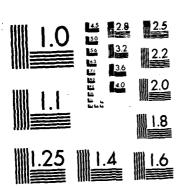
(STARS) WORKSHOP MARCH 24-27 1986(U) NAVAL RESEARCH LAB 174 UNCLASSIFIED F/G 12/5 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

laval Research Laboratory

ashington, DC 20375-5000

NRL Publication 0120-5150



AD-A198 120

MC FILE CUE

Software Technology for Adaptable Reliable Systems

(STARS) Workshop

March 24-27, 1986



Approved for public release; distribution unlimited.

Øb

Software Technology for Adaptable Reliable Systems

(STARS) Workshop

March 24-27, 1986

Accesi	on For	
DTIC	ounced []	
By Distrib	ution /	INSTACTED S
Į.	variability (Curry)	
Dist	Avai ana c. Sozola	-
A-1		

Naval Research Laboratory Washington, DC 20375-5000





CONTENTS

PREFACE	iv
VISE: A VISUAL SOFTWARE DEVELOPMENT ENVIRONMENT SUPPORTING REUSE	1
A PROTOTYPE SYSTEM TO AUTOMATE THE QUALIFICATION OF SOFTWARE FOR USE IN A REUSABLE SOFTWARE INVENTORY SYSTEM	5
INFORMATION THEORY AND SOFTWARE REUSE	37
WHY PROGRAMS BUILT FROM REUSABLE SOFTWARE SHOULD BE SINGLE PARADIGM Elaine N. Frankowski	51
UNDERSTANDING ADA (R) SOFTWARE REUSABILITY ISSUES FOR THE TRANSITION OF MISSION CRITICAL COMPUTER RESOURCE APPLICATIONS	59
COSMIC - NASA'S SOFTWARE DISTRIBUTION CENTER	87
THE DESIGN FOR REUSABLE SOFTWARE COMMONALITY	97
MSAT BRIEF - NARRATIVE TO ACCOMPANY THE HIGH LEVEL TECHNICAL BRIEF C. Ogden	133
A CLASSIFICATION SCHEME FOR REUSING SOFTWARE COMPONENTS	155
GUIDELINES FOR WRITING REUSABLE ADA (R) SOFTWARE	181
ALTERNATIVE TECHNOLOGIES FOR SOFTWARE REUSABILITY	195
CREATING REUSABLE ADA (R) SOFTWARE Ed Berard	205
AUTOMATED MEASUREMENT SYSTEM (AMS)	297
DOD-STD-2167 REV A PLANS	355



PREFACE

WORKSHOP ON APPLICATIONS SYSTEMS AND REUSABILITY 24-27 March 1986

The Department of Defense Software Technology for Adaptable Reliable Systems (STARS) Project is holding its third workshop on Applications Systems and Reusability 24-27 March 1986 at the Ramada Inn, Oxon Hill, MD.

The intent of this workshop series is to seek sources of information and expertise in the building of mission critical applications software. Reusable part specification, building, testing, maintaining and reutilization are of interest. With the promulgation of Ada as a single High Order Language (HOL) to build future applications within the three DOD services, there exist new opportunities of reuse of software. Reuse could reduce software system development time and maintenance costs, and improve reliability. The intent of the workshop will be to present and discuss summarized material on the following issues:

(1) Specification/Design:

(2) Reusable Component Definition;

√(3) Validation of Software Components;

9(4) Library Experience;

(5) Automated Part Composition,

16) Logistics of Organizational Reuse of Software and as Government Furnished Material, and Problems Encountered with Data Rights and License Arrangements, and User Liability Claims; 2(7) Encouraging Deposits, and (8) Ada Experience.



VISE: A VISUAL SOFTWARE DEVELOPMENT ENVIRONMENT SUPPORTING REUSE

Adarsh K. Arora James C. Ferrans Rob Gordon

Gould Research Center 40 Gould Center Rolling Meadows, IL 60008

Objectives

The Visual Software Development Environment (VISE) project is part of the Visual Programming Project at Gould Research Center (ARORA85). The overriding objectives of the Vise Project are to decrease software development time and improve programmer productivity through the use of workstation-based, graphical software development environments. Since software development is a long and complex process, we have chosen to develop tools to aid in the detailed design, coding, and debugging stages of the software life-cycle.

The major objectives of Vise are to:

- (1) Build an integrated, workstation-based software development environment that allows design, coding, testing, and debugging to occur in parallel without forcing the user to make expensive "context switches" between the editor, the compiler, the linkage editor, and the debugger;
- (2) Exploit advanced user interface technology to make programming more productive;
- (3) Develop rich libraries of reusable software components which can be intelligently searched for incorporating components into a developing program; and
- (4) Use design information to automatically generate target language code, and to select data structures and procedural templates.

The accomplishment of these goals would represent a considerable improvement in programmer productivity and would reduce the duration of the software life-cycle.

Description of Project

Current software development tools offer little help in mapping the initial abstract solution of a problem into a target language program. They generally address isolated stages of the life-cycle. For example, program design languages (e.g., (Teic77) are used to specify a high-level description of a program, while syntax-directed editors ease the coding task. No general tools exist that assist in the transition from a high-level problem description to target language source code

The first rough draft of a program is typically an informal, structured-English

description. Subsequent versions refine this description to successively lower and more precise decompositions of the problem until the target language is reached. Vise captures data types and procedural information from the later design stages and uses it to generate the target language code directly. To support this, Vise allows a program to contain pseudo-code statements called design notes as well as normal target language text. The programmer can select a design note and ask the system to refine it into target language code. After this is done, the refinement code is inserted and the design note becomes a comment introducing the code. Refinement is driven by information kept in one or more

Software Components Libraries (SCLs). SCL organization and refinement is discussed in a subsequent section.

Vise will also address testing and debugging. Graphical data structure display and animation and control flow tracing will be used to enhance the programmer's ability to monitor program behavior. Trace and single-step facilities will provide the capability of viewing how each program statement affects a particular data object.

Finally, a long term goal of the Vise project is to incorporate an expert system to aid in automatic data structure selection. After asking the user about usage patterns and access requirements, the assistant will suggest appropriate data types and give the rational behind their selection.

Technical Approach

Vise is being developed on Sun workstations using UNIX, C, and Objective C. With the decreasing cost of networked, single-user workstations, soon it will be practical to equip every programmer with one. The graphics capabilities of these workstations make them a logical choice for hosting a visual development environment. We are initially supporting the C language, but our technology is general and can be easily applied to ADA, FORTRAN, and other languages.

The Task Environment Manager (TEM) is the control and coordination layer packaging all Vise tools. It is responsible for tracking the environment of a programming task: which source files belong to it, which libraries it references, etc. It is used to import existing target language files into the system and export files out of Vise. In the C language implementation, the TEM export facility generates the "make" file used to build the task with conventional UNIX development tools.

A syntax-directed editor and an incremental compiler form the foundation of Vise. The programmer can have multiple edit windows open simultaneously on different source files, and every change he makes is compiled and checked as he makes it. The incremental compiler keeps an internal abstract syntax tree of the program in executable form. As in PECAN (Reis84), we intend to support geographical display and editing of programs (e.g., via flow charts and Nassi-Schneidermann diagrams) as well as textual editing. The target language syntax is augmented to accept design notes in arbitrary natural language.

The Animator/Debugger is an interpreter that walks the abstract syntax tree and executes the program. It supports the typical facilities of today's symbolic debuggers (e.g., control flow tracing, breakpoints, watchpoints, stepping) as well as higher-level graphical display of data structures. Control flow animation is done through the program editing views, as in PECAN.

Vise also contains a help facility and an Agenda Manager (AM). The AM is used to keep agendas of pending items that need to be done. For instance, if the editor finds a syntax error, this error is automatically added to an agenda for that source file, and automatically removed when it is fixed. The user can create his own agendas of reminders.

The final component of Vise is the Software Component Library Manager (SCLM). We describe its organization and how it supports design note refinement in the next section.

Software Component Library Manager

Software Component Libraries are collections of reusable software modules along with documentation and other supporting information. The SCLM manages these libraries and supports browsing and updating operations. Libraries may be organized in any manner, both topically and organizationally, and may be owned by anyone.

The major type of component in an SCL is the Abstract Data Type (ADT). This corresponds closely to an ADA package. Support also exists for generic ADTs, which correspond to ADA generic packages. ADTs may be derived from generic ADTs, and the user may create instances of ADTs in a process called instantiation.



SCLs also contain independent functions (those that do not pertain to an ADT) and code templates, along with a hierarchical index to them. Documentation, sources, objects, and preprocessor "include" files are also available for use or perusal. They also contain linguistic information needed in design note recognition.

When a user picks out a design note and asks the editor to refine it into target language code, the editor passes the note to the SCLM. The SCLM uses a simple keyword recognition scheme to search each SCL for matching actions, and the user selects which one looks most appropriate if more than one action was matched. The action refers to some independent function, code template, or ADT operation function, and the appropriate function call or other target language code is inserted into the program in place of the design note. The note becomes a comment to the new code.

The user can make refinement very loose (only one word in the design note matches a word in an action template) or very restrictive (the design note must match the action template semantics exactly). When the user picks the desired action, linguistic analysis is used to extract information from the design note and insert it into slots in the generated code.

STARS Program Relationship

The Vise Project addresses two major goals of the STARS program: It provides software life-cycle support as desired by the Software Engineering Environment (SEE) portion of STARS, and also meets the requirements for portable and reusable software parts as specified by the Applications segment.

An ADT-based library architecture is well-suited to the goal of similar applications, allows parametric refinement of parts, and supports parts composition. Libraries can run the gamut between generically useful and application—or user-specific. Software developers working on an application (e.g., signal processing, avionics, missile control, navigation) often use a particular model when designing systems. The availability of software parts corresponding to the model increases productivity.

The Visual Programming Project also has under its umbrella a domain specific

application of Vise technology. Gould Research Center has recently awarded a one million dollar contract from LABCOM to develop a visual VHDL Design Workbench for hardware designers. With the DoD requirement that all VHSIC chips be specified using VHDL, an Ada-like hardware design language, the DoD has initiated work on constructing a development environment for the VHDL designer. The VHDL Workbench provides a graphical environment for the specification of hardware components and the automatic generation of portions of VHDL through interaction with this environment.

Current Status

The development of Vise has been divided into three phases. In Phase I we are devising the core system: the text editor, the incremental compiler, the Software Component Library Manager, design note refinement the Task Manager, and the Agenda Manager. We have six people working on the project and expect to be finished by the summer of 1986.

In Phase II we intend to enhance the Phase I components and add the following capabilities: graphical program display and editing methods, new SCLM capabilities, additional SCLs, and the visual Animator/Debugger. This is scheduled for completion in 1987.

The last phase will explore expert systems technology for data structure selection and the use of better natural language recognition techniques in design note recognition.

References

(Aror85) Arora, A.K., Chan D., Ferrans, J.C., and Gordon, R. "An overview of the Vise visual software development environment," Proceedings IEEE Compsac85, Chicago, IL, 9-11 October 1985, pp. 464-471.

(Reis84) Reiss, S.P. "Graphical program development with PECAN program development systems," SIGPLAN Notices, 19, 5, (May 1984), pp. 30-41.

(Teic77) Teichroew, D. and Hershey, E.A., "PSL/PSA: a computer-aided technique for structured documentation and analysis of information processing systems," IEEE Transactions on Software Engineering, 3, 1, (January, 1977), pp. 41-48.

A PROTOTYPE SYSTEM TO AUTOMATE THE QUALIFICATION OF SOFTWARE FOR USE IN A REUSABLE SOFTWARE INVENTORY SYSTEM

F.C. Blumberg III C.T. Shotton, Jr. T. Zyla

Planning Research Corporation 1500 Planning Research Drive McLean, Virginia 22102 March 6, 1986

Abstract

In the software engineering environment of the 1980's, the reusability of software is expected to be a prime factor influencing the productivity of software development organizations. This paper discusses a prototype system developed at Planning Research Corporation used to demonstrate the feasibility of determining the reusability of software which was not originally designed for reuse.

Introduction

PERF

Under constant pressure to produce computerized systems of ever increasing size and complexity, most large organizations engaged in software development are forced to continuously find ways to improve productivity. In today's environment, a software development organization of any size usually controls thousands upon thousands of lines of diverse software, most of which was not explicitly designed for reuse. Significant time and cost savings can be realized in most organizations if an effective means to reuse this software can be developed. Clearly, some pieces of this software can, in practice, be reused in applications and systems not originally envisioned by their designers. The heart of the problem lies in finding an efficient way to identify and select the potentially reusable software.

The Prototype Project

In order to gain a better understanding of how to exploit the potential of this body of software, Planning Research Corporation decided to pursue the aevelopment of a prototype system to qualify software for use in a reusable software inventory system. During the development of the prototype, it was

shown through internal projects within the Productivity Products Group at Planning Research Corporation, that systems on the order of 30K+ lines of code can achieve reusability rates in excess of 60%, even if the reused software was not originally designed and implemented for reuse. Identifying and extracting the reused pieces of software from their original libraries was a labor intensive and time-consuming process which is intended to be automated by the prototype.

The development of the prototype qualification system has also demonstrated that the benefits of large-scale reuse can be magnified by integrating the software reuse qualification process into a SEE (software engineering environment). The interactive screens, user commands, flow of data, implementation techniques, and database storage techniques used in the prototype system are, by design, consistent with Planning Research Corporation's APCE (Automated Product Environment) Control product, corporation's standard software engineering environment. Extending the APCE to include an automated reusable software qualification process is a simple task which allows software reuse to occur in an organized and disciplined fashion.

At the beginning of the prototype development project, it was decided that the prototype should exhibit the following characteristics:

- o It should be able to qualify software not explicitly designed for reuse, as well as software explicitly designed for reuse.
- o It should not be tied to a specific computer language; it should be able to accept "raw" input software written in many different source languages. Although Ada is without a doubt the language of choice for future development, there is a large body of software written in other languages that can be effectively reused.
- o The qualification process should be adaptable so that the qualification criteria can be adjusted as more is learned about the characteristics of reusable software.
- o Software accepted into the reusable inventory by the qualification process should be classified and stored so that it can later be retrieved by its attributes, and it must be stored in a manner compatible and consistent with an organization's SEE or software factory system.

The prototype design consists of three major subsystems: the Analyzer Subsystem, the Qualification Subsystem, and the Interrogation Subsystem. Data flows through the prototype from the Analyzer Subsystem, where "raw" input software is inspected, to the Qualification Subsystem, which makes decisions based upon the analysis of the "raw" input software, to the Interrogation Subsystem, which is used to retrieve selected pieces of software stored in the reusable inventory.

The Analyzer Subsystem

A block diagram of the Analyzer Subsystem is presented in Figure 1. "Raw" input software can be analyzed whether or not it was explicitly designed for reuse. The Analyzer Subsystem determines the source language of the "raw" input software and, through the use of a grammar for the source language, develops metric data from the input. If the "raw" input is a large piece of software, for example an entire application program, the Analyzer Subsystem breaks down the input into smaller pieces and

records the relationships between the pieces.

Because the Analyzer Subsystem is constructed so that it is driven by a specific grammar file for the "raw" input software, it is flexible and can accommodate a wide variety of source languages. The prototype system has been demonstrated to be capable of processing both Ada PDL and PASCAL input software. The estimated level of effort required to extend the language processing capabilities of the prototype is on the order of a few man-weeks.

The Qualification Subsystem

At the time the prototype was developed, it was felt that the decision making process required to qualify "raw" input software for reuse was not fully understood. Casting the decision making process into a complex, and difficult to change program was not an acceptable or practical approach to the problem. It was obvious that the prototype had to include Al techniques to avoid recompiling or re-linking major pieces of the prototype for each change to the qualification decision making process.

Figure 2 is an overview of the flow of data through the prototype's Qualification Subsystem. Metric data, information about the relationships found in the "raw" input software, and some user-supplied text data are sent through a qualification process controlled by a Planning Research Corporation developed A1 module called GEM (Generic Expert Module).

GEM is used to apply a set of qualification rules against the metric and relational data supplied by the Analyzer Subsystem. GEM determines whether a piece of "raw" input software is qualified to be stored in the reusable software inventory. As the dynamics of qualifying "raw" input software come into better focus, the rule base can be easily modified with re-compiling or relinking any software in the entire prototype system.

Rules on the rule base file are written in straight-forward English-like constructs, and are interpreted dynamically at run-time by the GEM A1 module. GEM has powerful logical operators that allow rules to be developed that can accommodate a wide range of input software sizes and source languages. In addition to logical operators, GEM has computational features which allow a rule base to be developed that can calculate arbitrarily defined values such as portability factors, logical complexity factors, and interface complexity factors.

As an additional means to cope with the dynamics of its decision making process, the Qualification Subsystem has interactive screens which allow a user of the prototype to review decisions made as a result of applying the rule base to the Analyzer Subsystem's data. Numerical and relational data produced by the Analyzer Subsystem relative to any particular piece of "raw" input software can be displayed on command. 25 well qualification decisions made by GEM. A simple command from an interactive screen can be used to override any automated decision.

As pieces of "raw" input software are accepted by the Qualification Subsystem, they are stored in the reusable inventory along with numerical and relational data produced by the Analyzer Subsystem. The Interrogation Subsystem of the prototype can be used to select the stored pieces of software based upon various attributes, and can be used to display numerical and relational data.

Reuse and Development Standards

An unexpected use of the prototype system surfaced after its development. It was

realized that the prototype system's rule based could easily be modified to enable the prototype to be used as a fast and highly efficient IV&V tool. With a rule base that reflects an organization's fundamental software development standards (for example, number of lines per module, amount of comments, overall complexity per module, etc.) the prototype system could be employed to automate first level software inspections or walkthroughs which have traditionally been performed manually.

Conclusions

A number of things were learned during the development of the prototype.

- o Reusability in excess of 60% can be achieved, even when the reused software was not intended for reuse.
- o Large-scale software use is best carried out as an integrated function of a SEE.
- o A reusability system need not be tied to a specific source language
- o Some of the problems of automating software development standards inspection and identifying reusable software are similar.
- o Finally, Al tools and techniques can be applied in harmony with conventional software to attack ill-defined and complex problems frequently encountered in the development of automated software engineering environment systems.

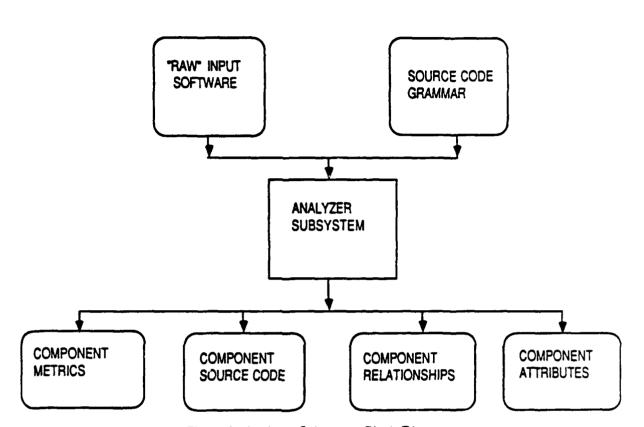


Figure 1. Analyzer Subsystem Block Diagram



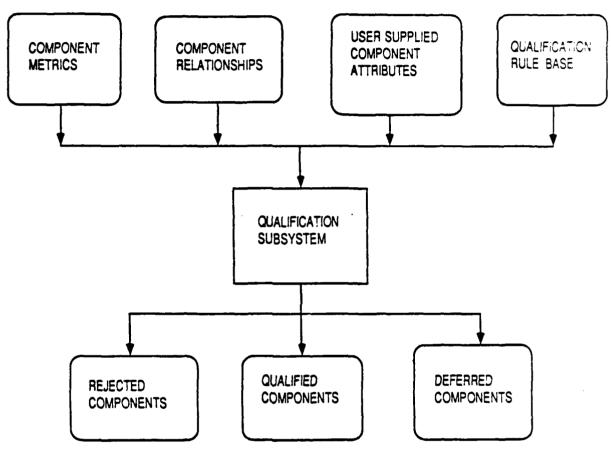


Figure 2. Qualification Subsystem Block Diagram

A Prototype System to Automate the Qualification of Software for Use in a Reusable Software Inventory System

March 1986



(j)

- Computerized systems continuing to increase in complexity
- Huge volumes of software under control of software development organizations
- Constant pressure to increase software development productivity

Experience With Internal Projects

- Reusability rates in excess of 60% have been achieved with software not originally designed and implemented for reuse
- Identifying and extracting reused pieces of software from their original libraries is labor-intensive and time-consuming

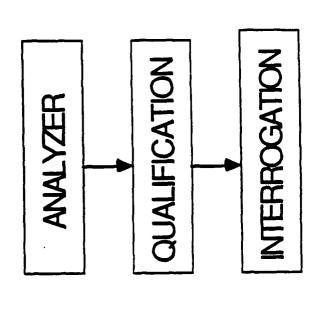
THE

Design Goals of the Prototype

- Ability to qualify software not explicitly designed for reuse
- Not based upon a specific source language
- Adaptable qualification decision making process which can be adjusted dynamically
- Software accepted by the prototype stored so that it can later be retrieved by attributes
- Compatible with APCE, PRC's standard software engineering environment

Subsystem Data Flow

W

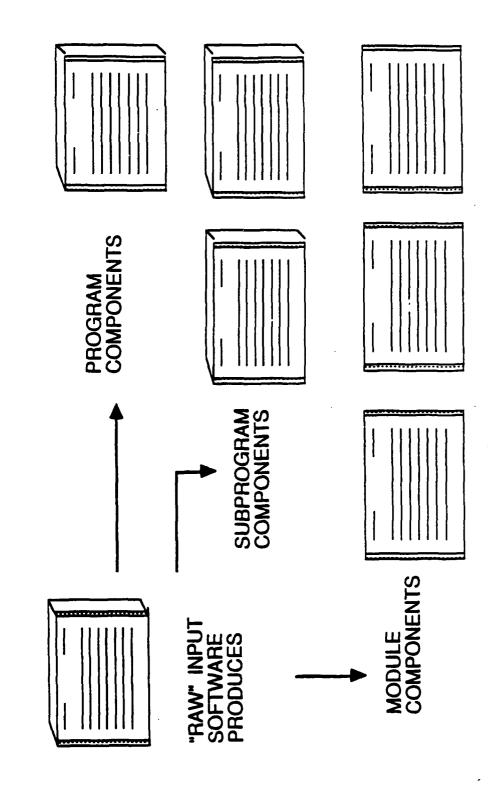


THE ANALYZER SUBSYSTEM

Analyzer Subsystem Overview

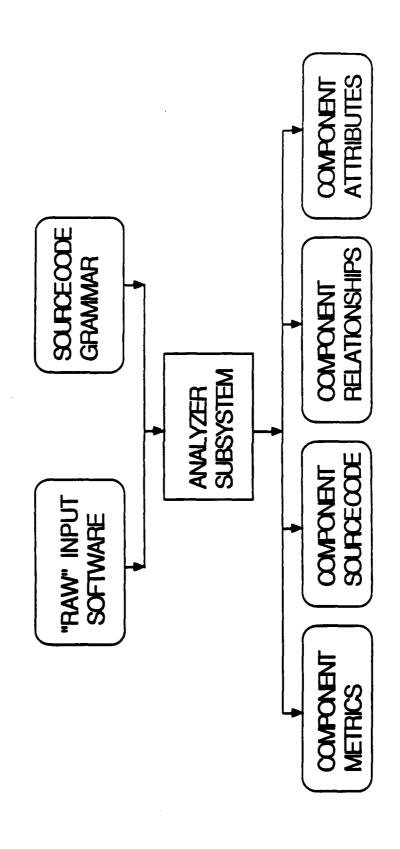
- Driven by grammar developed for the "raw" input source language
- Flexible subsystem can easily be extended by adding new grammars
- Has been demonstrated with:
- Ada PDL
- PASCAL

BREAKDOWN OF "RAW" INPUT SOFTWARE



Analyzer Subsystem Data Flow

 \mathcal{M}



ا ع

QUALIFICATION SUBSYSTEM





TO THE OWN OF THE PROPERTY OF

Qualification Subsystem Overview

- relational data from the Analyzer Subsystem to make Applies English-like rules against metric and qualification decisions
- Utilizes Planning Research Corporation's Generic Expert Module (GEM) Al software
- Catalogs qualified pieces of software
- Has interactive review screens and override commands available to prototype user

The Decision Making Process for Qualification

- Not fully understood at time of prototype development
- Best implemented using Al techniques
- Avoids developing complex and difficult to change programs
- Allows decision rules to be quickly changed as more is learned about qualifying software

22

Generic Expert Module Features

- Driven by rule base file written with straight-forward English-like constructs
- Dynamically interprets rules during program execution rules to not have to be compiled or linked into program
- Has powerful logical operators
- Has computational capabilities

Sample Rule Base

```
(COMPONENT'S NO_LOOPS_WITH_CONSTANTS * 100) / (COMPONENT'S NO_LOOPS + 1)
                                                                                                                                                                                                                      COMPONENT'S REUSABILITY IS NOW COMPONENT'S REUSABILITY + 10
                                                                                                                                                                                                                                                                                                                                                                COMPONENT'S LANG STAT IS ACCEPTED
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               COMPONENT'S STATUS IS NOW ACCEPTED
                                                                                                                                                                                                                                                                                                                                                                                                                                             COMPONENT'S REUSABILITY > 44
                                                                               COMPONENT'S TYPE IS PROGRAM
                                                                                                                                                                                                                                                                                                                                                                                                                    COMPONENT'S IF COMPLEXITY
                                                                                                                                                                                                                                                                                                                                                                                          COMPONENT'S PORTABILITY
                                                       ALL OF:
                                                                                                                                                                                                                                                                                                                                      ALL OF:
                                                                                                                                                                                              THEN:
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       THEN:
                                                                                                                                                                                                                                                                                                          II:
                             IF:
I
                                                                                                                                                                                                                                                                               X 1
```

QUALIFICATION RULE BASE COMPONENTS USERSUPPLIE **ATTRIBUTES** COMPONENT QUALIFICATION Qualification Subsystem Data Flow COMPONENTS SUBSYSTEM QUALIFIED RELATIONSHIPS COMPONENT COMPONENTS REJECTED CONFONEN METRICS

THE INTERROGATION SUBSYSTEM

800

21 March 1986

XX

AUTOMATED REUSABLE COMPONENT SYSTEM

MAIN SYSTEM MENU

09:15 HRS

1. ALYE Analyzer Subsystem 2. QUAL Qualification Subsystem

INTE Interrogation Subsystem

O. EXIT

First Wenu Choice.

BEREEFERSTANDE PROTOTYPES

Command:

2

09:19 HRS

21 March 1986

SYSTEM COMPONENT AUTOMATED REUSABLE

MENO SUBSYSTEM INTERROGATION

Display Components by Key KEY

Display Components by Attribute ATTR

*****TTCOF PROTOTYPE**

Enter Menu Choice.

Command: ATTR

(H)

(WY

21 March 1986

09:26 HRS

AUTOMATED REUSABLE COMPONENT SYSTEM
INTERROGATION SUBSYSTEM
SELECT BY ATTRIBUTE

###

SELECTION

Class: SUBPROGRAM

Language: PASCAL

Manufactuer: NAVELEX

Target Hardware: HONEYWELL L66 Operating System: GCOS-III

General Function:

PARSING

*ENTER - GO, BACK, QUIT, or EXIT

Command: GO

ည်ရ

21 March 1986

09:28 HRS

AUTOMATED REUSABLE COMPONENT SYSTEM
INTERROGATION SUBSYSTEM
SELECT BY ATTRIBUTE

*** SETESTED COMPONENTS ***

Description	AN ADD1	FIND SYM	SAW SYM	DUNG SYM	RULE APPLIES	SYNTHESIZE	NEXTCH	NEXTSYM	PARSE		
QI	090000	000064	90000	990000	690000	W90000	00000C	090000	00007B		

NEXT, TOP, KEY, SELE, QUIT, or EXIT *ENTER

Command: X 66

END OF SELECTED COMPONENTS

2

Language: PASCAL

Manufactuer: MAVELEX

Target Hardware: HONEYWELL L66 Operating System: GCOS-III

General Function:

PARSING

Description:

DUMP SYM

- ATTR, METR, RELA, LIST, MORE, QUIT, EXIT *ENTER

Command: METR

مح

21 March 1986

AUTOMATED REUSABLE COMPONENT SYSTEM

09:33 HRS

INTERROGATION SUBSYSTEM SELECT BY ATTRIBUTE

*** COMPONENT METRICS ***

Component ID: 000066 State: QUALIFIED

Class: SUBPROGRAM

Language: PASCAL

Metric Metric Metric Metric Metric Metric Metric \$ Array Rof w/ k: Loops with k: Freq of Use: Metric 1: Reusability: Portability: Interface Cmpx: Calling Params: Returns: Exits: Lines of Code: IF Statements: External Refs: Do Statements:

80

:01

Metric

Metric

Comments/Total:

9,02

ATTR, METR, RELA, LIST, MORE, QUIT, EXIT 1 *ENTER

Command: RELA

21 March 1986

09:34 HRS

AUTOMATED REUSABLE COMPONENT SYSTEM
INTERROGATION SUBSYSTEM
SELECT BY ATTRIBUTE

*** COKPONENT RELATIONSHIPS ***

Component ID: 000066
State: QUALIFIED

Class: SUBPROGRAM
Language: PASCAL

Referenced
Component Class Type Parameters of Code
000067 MODULE P 1 9

== * * * TTCOF - ATTR, METR, RELA, LIST, MORE, QUIT, EXIT *ENTER

PROTOTYPE**

Command: ATTR

Conclusions

- Reusability rates in excess of 60% can be achieved in some cases, even if the reused software was not designed for reuse
- Identifying reusable software is labor-intensive and is is best carried out as an integrated function of an automated SEE
- A reusability system need not be tied to a specific source language
- Some aspects of automating software reuse, enforcing software development standards, and automating IV&V functions are similar

Conclusions (continued)

with "conventional" software when attacking problems · Al tools and techniques can be applied in harmony in automated SEEs

INFORMATION THEORY AND SOFTWARE REUSE

Rodney M. Bond

ARINC Research Corporation 2551 Riva Road Annapolis, Maryland 21401

Abstract

In this paper a paradigm for software reusability currently being researched is reviewed. A general discussion of information theory is presented followed by a short discussion of software testing concepts. Dependency analysis is then presented as a possible approach to unifying these two fields. Possible paradigms for dependency analysis are then proposed along with some anticipated problems. Finally, a summary is given which identifies related fields to which this research might be applied.

Information Theory

Information theory research started in the 1920's in support of a need to model communication systems. The mathematical concepts used today were derived by the late 1940's by mathematicians including N. Wiener and C. Shannon. Information theory proposes to answer a question attributed to U.S. political scientist Harold D. Laswell(1), "Who says what to whom with what effect?" In attempting to answer this question, one activity included the development of a quantitative theory of information measure(2). Though not the only model, nor a universally accepted model, one theory measures "usefulness" of information based on three metrics: entropy, self-information, and probability. Entropy, H(•), in information theory is a measure of the uncertainty associated with a message source. Selfinformation, I(•) is a measure of the information contained in a particular variable, x; and probability is a measure of the chance occurrence of the i(th) variable, $p(x_i)$. These three measures are related by the equation:

$$E[1(x_i)] = H(X) =$$

$$\sum p(x_i)I(x_i) = -\sum p(x_i) \log (p(x_i))$$

where the summations are over "M' unique symbols, $E[\bullet]$ is the expected value function, and X is a random variable. When M = 2, where the symbols might be represented by [0,1], or [true,false], or some other mutually

exclusive and exhaustive pair of values, the equation has a maroman ximum value at $p(x_i) = 0.5$. Hence the maximum entropy, or unknown information, for a binary alphabet system occurs when there is an equal chance of something happening, e.g. being true or false.

Software Testing

During the fault isolation process of software testing an attempt is being made to obtain information through tests, specifically the identification of a "faulty" component. Obviously the meaning, amount, and scope of the available information has to be considered in order to achieve identification. If we limit ourselves to a binary scope of information, such that there are only two possible values for the information sought, i.e. an M 2 alphabet, we may arbitrarily choose the alphabet to consist of the symbols [good,bad]. We can now associate a meaning to these symbols. In this paper "good" and "bad" will be used to mean the result(s) of a test, with no other results possible. Now that the meaning and scope of the information to be gathered has been defined, the amount of the information to be gained from a test must be determined.

There is a significant variation in the amount of information that may be acquired from a test. Suppose that there is a software program with "X" inputs, "Y" modules, and "Z" outputs as components. We will also assume that the inputs must somehow be

generated, and the outputs evaluated. If all inputs, modules, and outputs were interconnected, an exhaustive test set would include on an order of magnitude X*Y*Z* tests. If we were to analyze this system to isolate an error, and assuming a single fault, we might start by testing an input. If the test was good, we would have very little information gain, only the knowledge that the one input was good. However, if the test was bad we would have isolated the error. Conversely, a good output test would tell us that all components feeding the output were good, whereas a bad output test would tell us only that one of the components feeding the output were bad. The information gain from the good output test is of more value since it identifies the status of more of the components.

Thus it appears that information gain is maximized with bad tests near the input and good test near the output. However, since the outcome of a test cannot be ascertained prior to the test, current strategies for determining the order in which the tests are to be performed include random selection, from outputs toward inputs, and from inputs toward outputs. Information theory can be used to generate a more reasonable strategy. From information theory we know that the maximum entropy occurs in our binary alphabet when the probabilities of "good" and "bad" are equal. Selection of the test with maximum entropy will provide an answer to the most "unknown" information possible from a single test. Hence our strategy should be to choose a test that gives the most nearly equal information gain, regardless of the outcome, eliminating the most entropy from the analysis. It can be shown that this strategy approaches the theoretical limit of eliminating half of the components from consideration with each test(3).

Dependency Analysis

In order to choose the test with maximum entropy some algorithm for the assignment of information gain must be established. One approach is a form of dependency modeling. For simplicity of example, assume we have four boolean variables: B1, B2, B3, and B4; and three software modules: M1, M2, and M3 related by the following

program segment.

Call M2 with B1 returning B3
--Routine M2
Call M3 returning B2
Call M4 with B3 returning B4

The following data and control dependencies can be identified. Variable B4 depends on module M4 and variable B3. Variable B3 depends on module M2 and variable B1. These are called first order dependencies. By inference through variable B3, we can also reason that B4 depends on module M2 and variable B1. This is called a higher order dependency. Through identifying all dependencies and applying a weighting algorithm, a figure for the information entropy value of each test, here represented by a boolean variable, can be determined. This would then provide values of entropy for our information theoretic approach to error analysis. For complex software topologies, computer processing will be required for determination of all higher order dependencies. The initial requirement would be the identification of first order dependencies, tests, and modules. This could easily be done manually, or possibly with a simple tool. The data may even exist already for systems developed with a structured analysis and design approach.

Paradigm Concepts

Now that we have identified the theory, one area of application, and an implementation approach to using information theoretics, the next step is to relate the presented material to the area of software reusability. The potential for application covers a wide spectrum of possibilities.

Reuse Metrics

If we just performed a dependency analysis of a software program on any one of several possible dependency relationships such as shared data, execution control, interfaces, scheduling, etc.; the analysis could be used to generate a relative metric associated with that characteristic. The metric is rela-

tive because it probably would have no absolute meaning, but would be very useful in characterizing the program. One program might have a shared data metric very high with respect to another program, both of which could be used as a reusable component within some other program.

Depending on the type and quantity of modifications required, the program with less shared data dependencies may prove easier to modify. Similar arguments could be generated for other types of metrics that might be generated.

Context-Free-Comparisons

The set of values which represent the dependencies within a program may also represent a fingerprint of that system. This fingerprint in general would be free of any context description of the program such as language of implementation, host computer, or field of application. Comparisons of these sets of data might be useful in identifying reusable but abstract concepts such as the requirements or design of a program.

E3 Methodology

Dependency analysis can be seen to have direct applicability to a Form-Fit-Function (F3) type of methodology (4). This is the use of "standard characteristics" for the specification of requirements. The characteristics could include any discussed, or even more complex characteristics based on advanced algorithms for determining metrics such as "coupling" or "cohesion." This is not equivalent to using a "black-box" description of a program because the dependency model includes the internal workings as well as the interfaces. To be a complete methodology. the "approach" not only would identify the best fitting software for reuse by these characteristics, possibly from a library of software; but would also provide insight into how or where modifications should be made in order to make two programs compatible.

Anticipated Problems

The concepts described in this paper are currently being applied by ARINC Research Corporation in the field of hardware system

testing(5). Research into the applicability of the methods for software testing is currently being funded internally. Some of the key issues to be addressed by this research are: 1) the identification of what dependencies can be generated, 2) the applicability of these dependencies to software related problems such as fault isolation, reusability, reliability, test development, fault tolerance, and security, 3) how the data acquisition for dependency analysis can be gathered, and 4) how can the problem be structured to minimize processing requirements.

Summary

The concepts of information entropy and self-information combined with probability have been applied to the field of hardware testing with considerable success. Research is being conducted into the extension of this work to the field of software testing. There is a potential for application of the basic concepts to many other fields of software and systems analysis. The primary consideration in our approach is that the problem must be formable as a dependency model.

Bibliography

- [1] Gordon, G.N. "Communication", Encyclopaedia Britannica, Vol. 16, p. 686, 1985
- [2] Shannon, C.E. "A Mathematical Theory of Communications", Bell System Technical Journal, G23-656,27, pp. 379-423, July 1948
- [3] Balaban, H. & Simpson, W. "Testability/Fault Isolation by Adaptive Strategy", Proceedings: Annual Reliability and Maintainability Symposium, Orlando, Florida, January 1983.
- [4] Boring, G. & Retterer, B. "Form, Fit, and Function Specifications", ARINC Research Corporation Technical Perspective Number 16, (1974).
- [5] Simpson, W.R. "STAMP Testability and Fault-Isolation Applications, 1981-1984", Proceedings; IEEE International Automatic Testing Conference, Uniondale, New York, October 1985.

Information Theory and Software Reuse

by

Rodney M. Bond

for presentation to

STARS Application Group Workshop

24-27 March 1986

Washington D. C.



<u>Abstract</u>

- Current Research
- Based on Information Theory
- Applied to Software Testability
- Uses Dependency Analysis

Information Theory

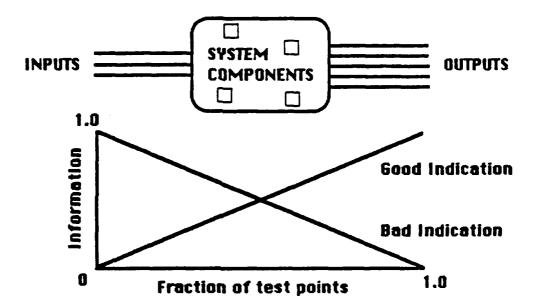
- "Who says what to whom with what effect?"
- C. E. Shannon
- Quantitative Theory of Information Measure
 - Not the only model
 - Not universally accepted
 - Uses three metrics
 - ●●● Entropy {H}
 - ••• Self-information {I}
 - ••• Probability {p}

•
$$H(X)=E[I(x_i)]=\sum_{j=1}^{M}p(x_j)*I(x_j)=-\sum_{j=1}^{M}p(x_j)*log[p(x_j)]$$

- •• E ≡ expected value function
- M ≡ number of unique symbols
- X ≡ random variable
- For binary system {M=2}
 - •• {0,1} {true,false} {nothing,something}
 - •• Mutually exclusive & exhaustive
 - •• $p(x_i)=0.5 \Rightarrow H(X)_{max}$
 - Equal chance ⇒ maximum entropy

Software Testing

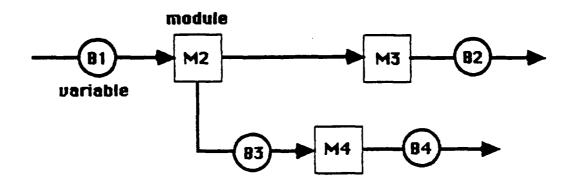
Let M=2, X={result of test}, x₁='good', x₂='bad'

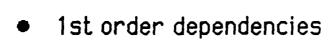


- Information gain maximuzed
 - good test near input
 - •• bad test near output
- Test strategies
 - No prior knowledge of test outcome
 - •• Random
 - Directed

- Information Theorectic
 - ●●● Select "equal information gain" test
 - Answers most "unknowns"
 - ••• Approaches half-interval limit

Dependency Analysis/Modeling





- Higher order dependencies & automation
- Weight to get Information Entropy test value
- Possibly use existing dependency data

Paradigm Concepts

- Reuse metrics
 - •• Data
 - •• Control ...
 - •• 'Characterizing" metrics
- Context-free comparisons
 - •• 'Fingerprint'
 - Applicable to abstractions

- F³ methodology
 - Use of standard characteristics
 - Not 'black box'
 - ID best fitting software from library
 - Provide modification data ...

Key Issues

- Dependencies?
- Dependency applicability to software problems
- Data acquisition
- Minimize processing requirements
- Is the hardware/software translation viable?

WHY PROGRAMS BUILT FROM REUSABLE SOFTWARE SHOULD BE SINGLE PARADIGM

Elaine N. Frankowski

Honeywell Computer Sciences Center 1000 Boone Avenue North Golden Valley, Minnesota 55427

Abstract

This paper argues that it becomes possible and economical to reuse software only when all the reusable parts exhibit a single paradigm and suggests that object-orientation is one recommendable paradigm.

Keywords: Ada, reusable software parts, object-oriented programming.

This research was supported in part by the Office of Naval Research under contract No. N00014-85-C-0666.

1. INTRODUCTION

Honeywell Computer Sciences Center's RaPIER (Rapid Prototyping to Investigate End-user Requirements) project is currently working on a methodology and automated support for constructing and using prototypes to investigate end-user requirements. Traditional requirements definition methods consistently fail to produce requirements from which satisfactory systems can be designed and built [ZAVE85]. We believe that rapidly built prototypes which model critical systems requirements can lead to early consensus on requirements that are acceptable to customers and feasible to implement.

The RaPIER approach [CSC86] is to build prototypes from reusable Ada software parts stored in a software database, and to express them in a very high level language that specifies how the parts are tailored and interconnected to form a complete prototype. The RaPIER project has the opportunity to test the feasibility of this approach with non-product software, in a non-time-critical milieu and to test different styles of implementating reusable part in this less demanding milieu. We intend to experiment with the approach using Ada on the Symbolics and a very high level prototype system

description language (PSDL) developed by International Software Systems Inc.(1) We chose the Symbolics(TM), a Lisp machine with support for dynamic linking, because prototyping demands a great deal of program modification at run-time. We expect their Ada compiler, which we will be receiving shortly, to exploit these facilities. We have already built two example prototypes using Lisp flavors as implemented on the Symbolics.

This paper is organized as follows: Section 2 discusses three assumptions about reusing software that underlie the recommendation of single-paradigm programs. Section 3 presents the reuse process and a critical requirement for reusable software. Section 4 discusses how single-paradigm programs facilitate the reuse process and support the critical requirements. Section 5 recommends object-orientation as a suitable paradigm. Section 6 discusses future work.

2. ASSUMPTIONS

Our experience in building prototypes by reusing Lisp flavors, and experience reported in [MATSUMOTO84, KER-NIGHAN84] leads us to make the following assumptions about the activity of software reuse and about reusable software parts.

- As a rule, reusable software, especially reusable code, will be modified each time it is used in a program. The simplest modification is the instantiation of generic parameters. More general modifications include enhancing a software part by adding features, restricting it by hiding features, and implementing it using features provided by others [GOGUEN86]. If software reuse is to be cost effective, modifications must be done systematically using "hooks" provided by the software part, and not simply by changing code.
- Reusable software parts, especially reusable code, must be built for reuse, either from scratch or by extensive retrofitting [MATSUMOTO84]. While it is possible to take code and "massage" it in order to reuse it, we claim that what is really being reused is a design, and that the massaging constitutes writing new code from a reused design. When a potentially reusable part is built, its author must consider the fact that it will be reused. This means, among other things, that the part should provide an appropriate abstraction, that is, behavior that is general enough to be useful in more than one program but specific enough so that there is not a large performance penalty for generality. The part should also provide appropriate hooks for systematic external modification. Other characteristics of reusable software are described in [STDENNIS86] which was produced by the RaPIER project.
- (3) Reuse will be most cost effective when reusers are familiar with the nature of the software parts that are available to them. This does not mean knowing a software part's behavior. No reuser can be familiar with the behavior of all the parts in a repository, although the locating process will be quicker if the reuser is familiar with the behavior of some candidate parts. What "familiar with the nature of the software parts" means is that the reuser understands "how things work." For example, Unix (TM) users understand that Unix utilities expect a

standard kind of input and output, and that piping can connect these utilities in a systematic way [KERNIGHAN84]. Unix users build programs out of reusable parts fairly easily because they understand "how things work."

3. THE REUSE PROCESS

There are four steps in reusing a software part, and one crucial requirement on reused parts.

Before a part is reused, it must be:

- o Located. A candidate for reuse must be found among all the reusable parts that are archived in some software database management system (SBMS). The SBMS must present users with a lucid classification scheme that appeals to their intuition. Each candidate part must be specified in such a way that the reuser is likely to find it.
- o Understood. Understanding a part means knowing what it does, how it does it, and how it can be reused. All these facts must be included in each part's specification. "What" is the part's function; "how" its operational behavior (for example, its reliability or performance); "how it can be reused" includes its expectations from its environment and the interface through which it is modified and incorporated into the program under development. Specifications can be natural language comments, formal specifications in a language such as the predicate calculus, operational specifications in a language such as Prolog, an Ada interface description, and so forth. Code as the only specification is unacceptable. Needing to read a part's code because of the poor quality of its specification is not desirable.

When the part is being reused, it must be

o Tailored. We assume that modifications will be needed. There are two kinds of modifications: {I} making new entities (types) from old by modifying something about the entity: for example, making a binary sort routine from a binary search routine by adding functionality to the search; or {2} making new instances of types: for example, instantiating an Ada generic with parameters that particularize it for the program in which it is included.

Changing code is the least desirable way to make a new entity or new instance. Code should be tailored from the outside using "hooks" such as parameters. [GOGUEN86] lists eight techniques for constructing new entities from old; none necessitates internal code modification.

o Connected. After a part is tailored, it is ready to be put together with the rest of the reusable parts that form the program under development. Connection requires build-time support in the form of a module interconnection language such as LIL [GOGUEN86] or PSDL [ISSI86]. Program construction is best accomplished in a development environment with module interconnection language-based tools. The RaPIER project is developing such an environment for constructing prototypes.

There are many requirements for reusable software parts [STDENNIS86]; one is crucial:

o Insulated from its Environment. A reused part must cause only the effects which constitute its documented behavior. It must not be a danger to the rest of the program by causing undocumented effects. It must be built not to interfere with any environment in which it finds itself. Conversely, a reused part must not allow the environment to endanger it. It must not make implicit assumptions about its environment that, when violated, will not allow it to function. It must not be open to uncontrolled modification of its internal state by its environment.

4. PROGRAMS BUILT FROM REUS-ABLE CODE SHOULD BE SINGLE PARADIGM

"A [programming] paradigm is a style of programming, supported by system facilities, that provides leverage in a range of programming tasks" [BOBROW85]. Some common paradigms and languages which support them are: procedure based (Ada), functional (Lisp), logic programming (Prolog), object-oriented (Smalltalk), and rule-oriented (OPS5). This paper argues that programs built from reusable software parts will have to exhibit a single paradigm.(1)(2) By implication, the reusable parts that compose such

programs will have to fit the paradigm.

We have concluded that programs built from reusable parts should exhibit a single paradigm at the top (interconnection) level. That is, the modules that are connected and the connections themselves should all be of a single style. Internally modules could be implemented in a variety of styles; however, the interfaces they present to the reuser should be semantically uniform.

One major argument for a single paradigm is that reusable parts that were not 'made for each other" must work together; that is, cooperate to achieve the system's goal while not interfering with each other. This is more likely to occur when all the reused parts are of a single paradigm: subroutines. Ada packages, flavors, and so forth. Another major argument for single paradigm parts is that all programs require run-time support and that simultaneous run-time support for multiple paradigms is not usually available. These same arguments also justify programs composed of reusable parts written in the same, multi-paradigm programming language. However, Ada is not a multi-paradigm language, and most new, sharable software will be developed in Ada. Therefore it is more practical to recommend a single paradigm for all the reusable parts in any program than to recommend several paradigms implemented in the same multi-paradigm language.

We now discuss how single-paradigm programming aids, to a greater or lesser degree, in each step of the reuse process presented in the previous section.

Locating. The major contributors to findable parts are good specifications and an SBMS with a perspicuous classification scheme. A single paradigm can help somewhat, in that classifying the same sort of entity is easier than trying to put functions, objects, logic routines, rules, etc. into the same In addition, classification scheme. library management tools can interact with standard components in a standard manner, allowing more possibilities for automation. For example, a tool could more easily produce explanations of parts' behavior by parsing the parts if

- the parts all follow the same semantic pattern.
- (2) Understanding. When users need to understand only one sort of thing, they accumulate background about that sort of thing. Thus they know basically how any part acts before investigating it and need only the added knowledge of precisely what it does. For example, understanding filters in general means that a user, encountering a filter, needs to ask only what the filter filters to have complete knowledge of how the filter behaves. When users need to understand many paradigms, they usually do not accumulate extensive knowledge about each.
- (3) Modifying. [GOGUEN86] lists eight kinds of modifications that will produce new entities from old. When programs are built from one type of entity, it is economical to invest in learning how to the hooks for external modifications of reusable parts in this paradigm. The hooks then make it easy for reusers to modify parts correctly. Even if, in extreme situations, code must be modified internally, the form of the code and its general behavior will be familiar, making it less likely that an internal modification will introduce errors into the code. When users write reusable code in many paradigms, they will not have learned patterns for "hooks," and so will make more provisions οf external modification. Reusers, in turn, will have to use less well thought out modification facilities that are also less familiar to them.
- Connecting. Two major benefits of single paradigm programming apply in this step. As mentioned above, programming with single-paradigm parts guarantees that the parts will fit together and that their run-time support can be provided. In addition, users will learn patterns for combining parts, thereby becoming more productive. The connection step should be supported by a program construction environment [GOGUEN86] that is based on a module interconnection language (MIL). If the module interconnection language is tuned to the paradigm, it

- can provide concise primitives for paradigm specific things such as message passing for objects. Thus users have to write less and, more importantly, to think less since the MIL's primitives obviate the need to built paradigm specific capabilities "by hand."
- (5) Insulation. There are paradigms such as the object-oriented paradigm discussed below that, through information hiding, provide insulation between parts and their environment in both direction. In addition, when all parts in a program follow any single paradigm, they are far less likely to interfere with each other. Multiple paradigms means different parts have different expectations of and behavior toward the environment, which can lead to interference.

5. THE OBJECT-ORIENTED PARA-DIGM

The RaPIER project has chosen the object-oriented paradigm for its reusable software parts. Object-oriented programming is the paradigm embodied in Smalltalk [GOLDBERG83] and the Lisp Flavor system [CANNON82]. The concept of an object as a named computational entity with identifiable behavior is central to object-oriented programming. An object's behavior is its reactions to the set of messages it "understands," where a message is a request to initiate processing or provide information, and "understanding" means call ... denotes an action, and sending a message ... makes a request [T]he interpretation of the message is left entirely up to its recipient." [RENTSCH82].

Object-oriented programming proceed top-down [BOOCH83] or bottom-up. Bottom-up object-oriented programming begins with a collection of reusable software objects such as the Smalltalk system's objects or a user's personal library. Objects for the problem at hand are built up by combining more primitive (system or user-defined) objects. Eventually the system contains the appropriate objects to solve the problem at hand. Then a program that uses these objects is written, often in a module interconnection language such as CMESA, PSDL or Bottom-up object-oriented LIL.

programming is a natural way to exploit a software repository's resources. The program under construction can certainly be designed top-down, but that design will take into account the available resources.

(1) This does not mean that every program must conform to the same paradigm, only that each individual program will be single-paradigm.

(2) Some problems should be solved by programs written in languages such as Loops [STEFIK86], that integrate multiple paradigms. A program in a multiparadigm language offers some of the same benefits we claim for singleparadigm programs.

programming Object-oriented specific benefits in the RaPIER setting. One important one has to do with the fact that a prototype is a vehicle for communicating about requirements between customers and product implementers. Traditional black-box requirements are difficult to discuss even among computer specialists, but especially between domain experts who are not computer scientists and the computer scientists who are solving their problems. [ZAVE85] that "An...important factor user/analyst communication is the ability of the user to grasp and evaluate the concepts behind any proposal. Experienced systems analysts report that an explicit operational model is much more helpful than black-box requirements....." That operational model has structure; it is the structure that facilitates discussion between customers and developers. We conjecture that users interacting with a prototype will view it as a collection of processes. autonomous, concurrent Although they will not think in computer science terms, of objects with local state and methods, and of asynchronous communication by message passing, they will think of a collection of processes, modules, or objects, each responsible for some part of the prototype's behavior. And although the objects from which the prototype is built will not be the same as the objects the user initially imagines, the builder can elucidate the

prototype's structure to the user, providing the structured communication vehicle recommended in [ZAVE85]. Another benefit of objects in the prototyping milieu is that they localize change, yielding an easily modifiable prototype. This idea is examined in detail below.

Objects aid the reuse process in the following ways:

(1) Locating. [BOOCH83a] motivates an object oriented design approach by pointing out that

"No matter what the particular application, the problem space is rooted somewhere in the real world ...in the problem space we have some real-world objects, each of which has a set of appropriate operations....

"Whenever we develop a software system, we ... model a real-world problem ... No matter what the implementation, our solution space parallels the problem space. ...the programmer abstracts the objects in the problem space and implements the abstraction in software."

We believe that both locating and understanding reusable parts is facilitated when the parts are the programmer's "natural" abstraction.

(2) Understanding. As stated above. objects are a natural model of the sort of software component that many programs should comprise. Thus people reusing them will have some intuitive understanding of "how they work" even before studying the paradigm. Objects (rather than subroutines, data structures, or general code fragments) are also an appropriate unit to understand in detail. They present complete enough behavior to be understood and used as units rather than as incomplete fragments, and to be combined without internal modification. The work on [LISKOV75], abstract data types Smalltalk [GOLDBERG83], and Flavors [CANNON82] bears this out.

The object's interface is the set of messages it can handle; each method can be

specified separately. This is a clean, simple interface: it is specified in small enough chunks to be easily grasped; it describes operations (methods), a notion that the user is already intuitively comfortable with. Thus the total specification presents a complete abstraction in easily understandable chunks.

(3) Modifying. An object is a complete unit of behavior; if it was built for reuse it presents an "appropriately useful abstraction" (see Assumption B), and thus the methods it provides will not have to be changed. Therefore, modifying an object will mean enhancing or restricting its behavior; both can be done from the outside by adding or deleting methods respectively. It is good software engineering to consider all changes to be either enhancements or restrictions, and to simply disallow internal changes to code. This is possible under Assumption B.

The object is a well-understood concept; reusers who modify it will know its pattern, and be able to modify it in the pattern. A module interconnection language can provide primitives for restricting an object's interface. For enhancing the interface, reusable objects can include some hidden methods (that is, methods not available to client software) that can serve as primitives for creating new methods. Hidden methods ensure modifications are correct in that they present a modifier with the same sort of "safe" interface that they provide for client software. Hidden methods are one of the investments that can be made when writing software for reuse.

(4) Connecting. An object-oriented program is, in concept, a loosely coupled collection of autonomous, concurrently active objects which communicate by message passing. Each object controls its own processing by interpreting the messages it receives and deciding how to handle each one based on its state and methods. This model has undemanding connection requirements: the module interconnection step must

only establish "wires" for messages to flow across, and provide some means of kicking-off the system. If all objects name the targets of the messages they send, interconnection can be totally automated. The model does require run-time support for message passing.

(5) Insulation from the Environment. Objects provide information hiding, not just modularity. No object can manipulate another's state except through well defined interfaces, the methods; objects control their processing by interpreting messages. Thus the likelihood of the environment spoiling an object's state is vanishingly small. By filtering their requests, objects do not allow interference. Because each object protects itself, interference is prevented in both directions.

6. FUTURE WORK

In order to make the object-oriented paradigm work in our RaPIER prototyping environment, we are investigating these questions:

- o What is an adequate implementation of an object/message passing model in an Ada based prototyping environment.
- o What features of the object model can be implemented in Ada? We will learn to make Ada parts that have as many of the characteristics of Smalltalk-like objects as Ada can support and learn how to do without the characteristics Ada cannot support. In particular, we shall investigate how to implement inheritance sufficiently well to obtain the time and effort savings from making a new object out of operations and data structures inherited from parent objects.
- o What are the build-time capabilities needed to support program construction by Ada object connection?
- o What are the run-time capabilities needed to support a system of Ada objects?
- o What kinds of modifications [GOGUEN86] are necessary to reuse objects



and how do Ada objects have to be constructed to allow these modifications to be made externally?

7. ACKNOWLEDGMENT

The author acknowledges many extremely enlightening discussions with Curtis Abraham of Honeywell's Computer Sciences Center who has designed and implemented two RaPIER prototypes. The ideas in this paper have benefited greatly from his insights into reusability and Lisp Flavors. He also read a draft of this paper and recommended several useful changes to it.

BIBLIOGRAPHY

[BOBROW85] Daniel G. Bobrow. "If Prolog is the Answer, What is the Question? or What it Takes to Support AI Programming Paradigms," IEEE Transactions on Software Engineering, Vol. SE-11, No. 11, November 1985, pp. 1401-1408.

[BOOCH83] Grady Booch. "Object-oriented Design," Tutorial on Software Design Techniques. Ed. P. Freeman and A. Wasserman, 4th edition (Catalog Number EHO205-5), IEEE Computer Society Press, 1983.

[BOOCH83a] Grady Booch. "Software Engineering with Ada, The Benjamin/Cummings Publishing Company, Inc., 1983.

[CANNON82] Howard L. Cannon. "A Non-hierarchical Approach to Object-oriented Programming," M.I.T. Technical Report (Draft), 1982.

[CSC86] Honeywell Computer Sciences Center. "Final Scientific Report to The Office of Naval Research: Joint Program on Rapid Prototyping," Honeywell Computer Sciences Center Technical Report, CSC-86-3:8213, March 1986.

[GOGUEN86] Joseph A. Goguen. "Reusing and Interconnecting Software Components," IEEE Computer, Vol. 19, No. 2, February 1986, pp. 16-28.

[GOLDBERG83] Adele Goldberg, D. Robson. SMALLTALK-80: The Language and Its Implementation, Addison-Wesley, Reading, MA, 1983.

[ISSI86] International Software Systems Inc. "Prototype System Description Language: Draft," private communication, January 1986.

[KERNIGHAN84] Brian W. Kernighan. "The Unix System and Software Reusability," IEEE Transactions on Software Engineering, Vol. SE-10 No. 5, September 1984, pp. 513-518.

[LISKOV75] Barbara H. Liskov, Stephen N. Zilles. "Specification Techniques for Data Abstractions," IEEE Transactions on Software Engineering, Vol. SE-1. No. 1, March 1975, pp. 7-19.

[MATSUMOTO84] Yoshihiro Matsumoto. "Some Experiences in Promoting Reusable Software: Presentation in Higher Abstract Levels," IEEE Transasctions on Software Engineering, Vol. SE-10 No. 5, September 1984, pp. 502-513.

[RENTSCH82] Tom Rentsch. "Object Oriented Programming," ACM Sigplan Notices, Vol. 17, No. 9, September 1982.

[STDENNIS86] Richard St. Dennis. "A Guidebook for Writing Reusable Source Code in Ada(R): Version 1.0," Honeywell Computer Sciences Center Technical Report, CSC-86-3:8213, Honeywell Computer Sciences Center, 1000 Boone Avenue, Minneapolis MN 55427, March 1986.

[STEFIK86] Mark J. Stefik, Daniel G. Bobrow, and Kenneth M. Kahn. "Integrating Access-Oriented Programming into a Multiparadigm Environment," IEEE Software, Vol. 3, No. 1, January 1986, pp. 10-18.

[ZAVE85] Pamela Zave. "The Operational Versus The Conventional Approach to Software Development," Communications of the ACM, Vol. 27 No. 2, February 1984, pp. 104-118.

UNDERSTANDING ADA (R) SOFTWARE REUSABILITY ISSUES FOR THE TRANSITION OF MISSION CRITICAL COMPUTER RESOURCE APPLICATIONS

A. Gargaro Computer Sciences Corporation Moorestown, NJ 08057

T. Pappas
Computer Sciences Corporation
Moorestown, NJ 08057

ABSTRACT

This paper identifies fundamental issues relevant to the successful reuse of Ada software in Mission Critical Computer Resource (MCCR) applications. The reusability of an Ada program is defined in the context of three criteria for evaluating the degree to which Ada software is reusable. These criteria are important to writing reusable software for the timely transition of MCCR systems to the Ada Language.

Ada (R) is a registered trademark of the U.S. Government Ada Joint Program Office

Prologue

A central idea in the design of the Ada language (Department of Defense 1983) is to assemble a program from independently produced software components. Therefore, the reusability of Ada software components (STARS 1985) is viewed as the cornerstone in reducing the cost of developing Mission Critical Computer Resource (MCCR) systems. If the promise of reusing Ada software components is fulfilled, the reduction in cost is expected to be significant (Anderson 1985)

There is little practical experience in reusing Ada software components for MCCR applications. In the initial transitions to the Ada language the reuse of software components may be adversely affected by fundamental issues that affect the writing of reusable components. Understanding these issues is necessary to managing the transition if the potential costs and benefits of Ada software reusability are to be predicted.

Approach

Several studies have reported on transitioning currently deployed MCCR systems to the Ada language (Friedman 1985). These studies have focused on evaluating the adequacy of the Ada language to meet existing performance efficiency requirements and do not specifically consider the reuse of transitioned software among different MCCR applications.

The results from the studies indicate that in transitioning to the Ada language, rigid performance requirements upon the run-time environment will necessitate the use of Ada constructs where their level of abstraction may be comprised by explicit and implicit dependencies upon the run-time environment. Consequently, developing Ada software that is both reusable and meets the performance requirements of MCCR applications presents a conflict. The conflict is exacerbated by programming practices that have exploited idiosyncrasies of the execution

environment. These practices have resulted in application specific techniques that are efficient but reduce the level of abstraction essential for software reuse.

For example, one requirement that pervades MCCR applications is the facility for periodic control of both concurrent and serial processing. Traditionally this requirement has been satisfied by variations of the Cyclic Executive which has become the classical paradigm for examining the efficacy of using the Ada language for real-time programming (Hood 1985; MacLaren 1980; Phillips & Stevenson 1984). Often the adaptation of the Cyclic Executive to provide efficient use of processing resources can lead to dependencies by the application software on programming techniques that are nonreusable. These techniques may persist after the transition depending upon the implementation of the Ada Run-Time System (RTS). In understanding the issues of software reuse, the ramifications of such techniques must be understood to perform tradeoff analysis between efficiency and reuse when transitioning to the Ada language.

To understand the reuse of Ada software components an approach must address, at a minimum, the issues of writing efficient code that is reusable in different run-time environments. Particular emphasis should be given to: performance efficiency requirements of MCCR applications as they affect software reuse, program composition features of the Ada language that facilitate the creation and use of reusable components, and the implementation options of the Ada RTS that may compromise software reuse. In this paper, the technical foci is directed towards the latter two topics.

ADA Software Reusability

Software reusability comprises the concept to execute a program in an execution environment different from that in which it was originally developed, i.e., transportability, and the concept to combine components from different programs in the development of a new program, i.e., reusability. The comprehensive support of the Ada language for modern software engineering principles, viz., abstraction, composition, encapsulation, and instantiation, provide a framework for writing reusable software. The distinction

made in this paper between the concepts of reusability and transportability of Ada software is discussed in the following paragraphs. This distinction partially resolves the inherent ambiguity of these two concepts and is consistent with the notion of both reusability "in the large" and "in the small" (Lubars 1986).

Program Transportability

The transportability of an Ada program is defined as the ability of a program to complete functionally equivalent execution in different environments consistent with the Ada language. Transportability is measured by the degree this execution can be achieved without modifying the source code. This definition is derived from an earlier one (Oberndorf et al 1982) and work that has been previously reported (Nissen & Wallis 1984; Pappas 1985). The stipulation for equivalent execution rather than identical execution recognizes that the processing capacity of the execution environment and the sophistication of the compiling system may affect the execution behavior of the program within the semantics of the Ada Reference Manual (RM) (Volz et al 1986). For example, the number of times a loop body is performed may vary because the source code invites compiler optimization. In addition, it does not exclude the use of representation specifications to influence execution since their use is perceived as essential to most MCCR applications.

Program Reusability

The reusability of an Ada program is defined as the ability of one or more of its components to execute with identical functionality in the construction of a new program. Reusability is measured in the degree that different components of the program can be used to construct new programs in the same and different execution environments. This definition is more stringent than the one recently proposed for developing reusability guidelines (Braun et al 1985) since three important criteria for evaluating program reusability are mandated: the transportability of the program, the orthogonality, i.e., functional independence, of its composition, and its freedom from dependencies on a specific implementation of the Ada Run-Time System (RTS). The definition does not

discriminate between writing reusable components and programs where their constituent components can be reused.

A necessary first step to reusing components in different execution environments is to achieve the transportability of the program. When only the program is to be reused, the distinction between reusability and transportability is the fidelity of execution, i.e., equivalent or identical. When a component is to be reused in different programs, e.g., an Ada generic unit, the transportability criteria ensures a context for validating execution.

Composition Orthogonality

In discussing composition orthogonality, it is convenient to introduce degrees of reusability. A component whose potential for reuse is low is said to be weakly reusable, while a component whose potential for reuse is high is said to be strongly reusable. These represent the extremes of reusability. Source modifications and limited applicability are expected with weak reusability, while with strong reusability no source modifications and potentially frequent applicability are expected. An effectively reusable component differs from a strongly reusable component only in that some source modifications may be required due to Ada language rules. In practice weak reusability is to be avoided, strong reusability strived for, with effective reusability actually obtained.

The orthogonality of a program's composition is an attribute of the program which reflects the independence of its components from the enclosing context. The stronger a component's dependence on its context, the less likely its potential for rouse since more of the context must be transported with it, i.e., weak reusability is more likely. Conversely, the weaker a component's dependence on its context, the greater the potential for the component's reuse since little, if any, of the surrounding context need be transported with it, i.e., strong reusability is more likely. When coupled with programming for generality, striving for context independence will yield effectively reusable, if not strongly reusable, software components.

Composition orthogonality is not an issue in program transportability since the entire context of each program component is transported to the new execution environment. It is only when a component is extracted from its context that composition orthogonality becomes an issue. The exception to this is a program whose main subprogram has parameters. But in this situation, the context dependency is on the execution context and not the application context. Therefore, the issue is one of transportability rather than reusability.

Degrees of reusability are illustrated in Example 1, where two versions of a binary search are shown. Example 1.a, which is typical of binary searches used in practice, is weakly reusable for several reasons. First, it has several context dependencies. Reuse of this example requires providing three entities in the new context: a named number, Max Table Elements, a type named Element Type, and an array named Table with the structure shown. If these entity names or the array structure are not appropriate in the new context, then the component must be modified. A second problem with this example is its lack of generality. In addition to only providing a binary search for a particular array, it strongly depends on the array index subtype being a subtype of Positive. This dependency is explicit in the Mid Point calculation and in the calculations of the left and right end points. The dependency is implicit in the use of zero to indicate that the element is not found in the Table. The result subtype of the Binary_Search function, Natural, extends the array index subtype by one value to allow it to serve a dual purpose -- return the array index upon a successful search and indicate failure upon an unsuccessful search.

Example 1.b illustrates a strongly reusable version of the binary search. Here, the function has been encapsulated within a generic package. Through the use of generic formal parameters, all context dependencies have been removed. In addition, the parameterization in Example 1.b encompasses all possible generalizations of this binary search that do not change its functionality.

Example 1.A - Weak Reusability

```
Table : array (1 .. Max_Table_Elements) of Element_Type;
function Binary_Search (Element : in Element_Type) return Natural is
   Left_Point : Positive := 1;
   Right_Point : Positive := Max_Table_Elements;
   Mid_Point
             : Positive;
begin
   while Left_Point <= Right_Point loop
      Mid_Point := (Left_Point + Right_Point) / 2;
      if Element < Table (Mid_Point) then</pre>
         Right_Point := Mid_Point - 1;
      elsif Table (Mid_Point) < Element then
         Left_Point := Mid_Point + 1;
         return Mid_Point;
      end if;
   end loop;
   return 0;
end Binary_Search;
```

Example 1.B - Strong Reusability

```
generic
   type Element_Type is private;
   type Index_Type
                    is (<>);
   type Table_Type
                    is array (Index_Type range <>) of Element_Type;
   with function "<" (Left, Right: Element_Type) return Boolean is <>;
package Binary_Search_Package_Template is
   function Binary_Search (Table: Table_Type; Element: Element_Type)
                           return Index_Type;
   Not_Found : exception;
end Binary_Search_Package_Template;
package body Binary_Search_Package_Template is
   function Binary_Search (Table: Table_Type; Element: Element_Type)
                           return Index_Type is
      Left_End : Index_Type := Table'First;
      Right_End : Index_Type := Table'Last;
      Mid_Point : Index_Type;
   begin
      if Table'Last < Table'First then</pre>
         raise Not_Found;
      else
         while Left_End < Right_End loop</pre>
            Mid_Point := Index_Type'Val (Index_Type'Pos (Left_End)
                                        + Index_Type'Pos (Right_End) / 2);
            if Element < Table (Mid_Point) then
               Right_End := Index_Type'Pred (Mid_Point);
            elsif Table (Mid_Point) < Element then
               Left_End := Index_Type'Succ (Mid_Point);
               return Mid_Point;
            end if:
         end loop:
         if Left_End = Right_End and then Element = Table (Left_End) then
            return Left_End;
         olso
           raise Not_Found;
         end if;
      end if:
   end Binary_Search;
end Binary_Search_Package_Template;
```

While there is no difference between effective reusability and strong reusability in Example 1.b, there are situations where a difference may occur. For example, consider a generic subprogram implementing a numerical algorithm such that the algorithm requires a real type. The "real" type is a generic formal parameter of the generic subprogram. If only standard mathematical operations are required for this type, then a private type can be used. The mathematical operations would be generic formal function parameters, with appropriate defaults, to the generic subprogram. If, however, accuracy demands necessitate the use of floating point or fixed point attributes, then two versions of the generic subprogram are needed: one for floating point types and one for fixed point types. In this case there is a difference between effective reusability and strong reusability. Both versions are effectively reusable but neither is strongly reusable.

One strongly reusable version could be written that would necessitate using a private "real" type. Several additional generic formal subprograms would need to be included as generic parameters, but rather than providing the user with any real benefit, these subprograms would simply serve to isolate floating point and fixed point attribute dependencies, perform type conversions, etc. While this version might satisfy the strong reusability notion of this paper, in reality, users would not be likely to use a generic component requiring generic actual parameters merely to comply with Ada's language rules.

Components that are effectively or strongly reusable seem to be consistent with good programming style so, ideally, all program components should be written in this manner. This would maximize the reusability of the program's components. In reality, this is not likely to occur since MCCR performance issues may dictate otherwise. While the binary search in Example 1.b may be strongly reusable, program tuning may require a weakly reusable version. In particular, the distributed binary search due to Knuth (Bentley 1982) may be needed in the tuned program. Since the distributed search could be produced by a program generator it may still be correct to view it as strongly reusable, but at the level of a program generator.

RTS Dependencies

The potential for RTS dependencies to affect the reuse of Ada program units can be appreciated by reviewing a specific example that presents a dependency on a particular implementation of task scheduling. This dependency does not necessarily prevent program execution from meeting the transportability criterion when the dependency is not satisfied in the environment to which the program is transported for reuse. However, successful reuse of the program unit that includes the dependency cannot guaranteed in the new environment.

The example is contrived to expedite a straightforward discussion and the referenced code does not represent recommended use of the language or a dependency that cannot be mitigated in some other way. The example originated from a revision to a program from the Ada Fair benchmark suite (Bardin et al 1985). The original program included packages designed to control access to a shared variable as a means of evaluating the integrity of the task scheduler. In the revised version, the access control task has been modified to service concurrent reader and writer tasks where the access protocol is biased in favor of writer tasks to simulate real-time updating of the shared variable. The shared variable is of a composite type and may be read concurrently by more than one task providing no task has been granted write access. Furthermore, writing must be serialized and outstanding writes should be serviced before a task is granted read access, since writer tasks are assigned highest priority.

The two code fragments to be examined are shown in Example 2. The first fragment is the select statement enclosed by the task that grants read/write access. The second fragment is the timed entry statement enclosed by the procedure that is called by the writer tasks. The dependency is associated with the use of the COUNT attribute in the iteration scheme of the while-loop that is designed to service all outstanding write requests before a new read is accepted.

The RM cautions against the use of the COUNT attribute because its value is not stable. In this instance sufficient stability is only required to ensure that the Start Write

Example 2 - Implicit RTS Dependence

(10

```
-- Task controlling read/write access to shared variable
   task body Rv_Control is
   select
   -- Activate new reader if no writer is waiting
    when Start_Write'Count = 0 =>
          accept Start_Read;
          Active_Readers := Active_Readers + 1;
   Or
   -- Activate writer if no active readers
    when Active Readers
                            = 0 =>
          accept Start Write;
          accept Stop_Write;
   -- Wait for active read to complete
          accept Stop_Read;
          Active_Readers := Active_Readers - 1;
          If Active_Readers = 0 then
   -- Activate and serialize waiting writers
             while Start_Write'Count > 0 loop
   -- >>> Implicit dependency on stability of COUNT
                   accept Start_Write;
                   accept Stop_Write;
             end loop;
          end if;
   or
      terminate;
   end select;
   end Rw_Control;
-- procedure called by writer tasks
   . . .
   select
      Rw_Control.Start_Write;
   -- Update shared variable with actual parameter from call
      delay Write_Time_Limit;
      Rw Control.Start_Write;
   -- Update shared variable to indicate that the writer was late
   end select;
```

entry queue is not decremented prior to accepting the Start_Write entry. This depends upon a class of First-In-First-Out (FIFO) task scheduling that prevents interruption of control task execution until it is blocked by the Stop_Write entry even in the presence of an expired timed entry statement. The dependency requires that expiration of the delay does not result in run-time action, viz., changing the state of the delayed writer task, until the executing task is blocked and a new task has to be executed.

This dependency does not preclude successful execution in a different environment where task scheduling is not guaranteed to maintain the stability of the value of the COUNT attribute. For instance, an RTS that implements a preemptive class of task scheduling may result in the value being decremented after the evaluation of the while-loop but prior to accepting the Start Write entry. However, because of the priority of the writer tasks and the Start_Write entry statement following the expired delay, the number of queued requests cannot decrease. Consequently, program transportability is achieved since execution is functionally equivalent in both environments.

When the above implicit dependency is not clearly stipulated, the control task may be mistakenly considered to be strongly reusable in the new environment on the basis of program transportability. An attempt to reuse the control task with a different procedure for writer tasks can have aberrant execution behavior in an environment that does not guarantee the stability of the COUNT attribute. A simple change to the timed entry statement that removes the Start_Write following the delay can cause the entry queue count to reach zero. The control task is now forced to unexpectedly wait at the Start_Write resulting in disruption to performance since the reader tasks are dependent for execution on a write request. This is contrary to the guard specification of the enclosing select statement. In a worst case situation, when no further writes are requested, the control task is blocked indefinitely from execution.

Epilogue

This paper has presented a refinement to the concept of reusability. This

refinement provides insight into understanding issues in writing reusable Ada software components for MCCR applications. Composition orthogonality and independence from the Ada RTS implementation are identified as useful criteria for assessing program reusability. Understanding these criteria will allow varying degrees of program reusability to be specified in transitioning MCCR applications to the Ada language. Composition orthogonality is important because many Ada features that facilitate program reusability have been avoided or unavailable in past MCCR application software that have commonly relied upon simple constructs with predictable performance efficiency (Bassman et al 1985). In addition, dependencies on the implementation of the Ada RTS to imitate low-level control of processing resources can thwart strong reusability achieved through composition orthogonality.

In managing the transition, software reuse should be safeguarded by balancing program reusability with performance during the design phase. Furthermore, reusable Ada software components will be facilitated by language implementations that are guided by the specification of classes of Ada Virtual Machines for MCCR applications and practical restrictions on Appendix F of the Ada RM. This would increase the likelihood of formally certifying the degree of reuse for software components (Cohen 1985).

References

Anderson, C.M. (1985) Reusable Software - A Mission Critical Case Study. AIAA Computers in Aerospace V Conference, pp 136-139.

Bardin, B. et al (1985) Report on the L.A. AdaTEC Ada Fair'84': Compiler Test Results. ACM SIGAda Ada Letters 4, No. 4, pp 52-58.

Bassman, M.J. et al (1985) Evaluating the Performance Efficiency of Ada Compilers. ACM DC SIGAda Washington Ada Symposium.

Bentley, J. L. (1982) Writing Efficient Programs: Prentice-Hall.

Braun, C. et al (1985) Ada Reusability Guidelines. SofTech Inc., 3285-2-208/2.

Cohen, N. H. (1985) Verified Ada: A Key to Reliable Software. AIAA Computers in Aerospace V Conference.

Department of Defense (1983) Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A.

Friedman, F. (1985) Issues Affecting Software Productivity due to the Introduction of Ada. Computer Sciences Corp., TR No. SP-IRD 4.

Hood P. (1985) Cyclic Executives: Pros, Cons, and Relation to Ada. SofTech Inc., Working Paper 1123-WP1.

Lubars, M. D. (1986) Code Reusability in the Large versus Code Reusability in the Small. ACM SIGSOFT SEN 11, No. 1, pp 21-28.

MacLaren, L. (1980) Evolving Toward Ada in Real-Time Systems. ACM SIGPLAN Notices 15, No. 11, pp 146-155.

Nissen, J. & Wallis, P. (1984) Portability and Style in Ada: Cambridge University Press.

Oberndorf, P. et al (1982) KAPSE Interface Team: Public Report Vol. 1. Naval Ocean Systems Center Technical Document 509.

Pappas, T. (1985) Ada Portability Guidelines, SofTech Inc., ESD-TR-85-141.

Philips, S. & Stevenson, P. (1984) The Role of Ada in Real-Time Embedded Applications. ACM SIGAda Ada Letters 3, No. 4, pp 99-111.

STARS (1985) STARS Workshop on Reusable Components of Application Software. Naval Research Laboratory.

Volz, R. et al (1986) Toward Real-Time Performance Benchmarks for Ada. University of Michigan.

STARS WORKSHOP

UNDERSTANDING Ada[®] SOFTWARE REUSABILITY ISSUES FOR THE TRANSITION OF MISSION CRITICAL COMPUTER RESOURCE APPLICATIONS

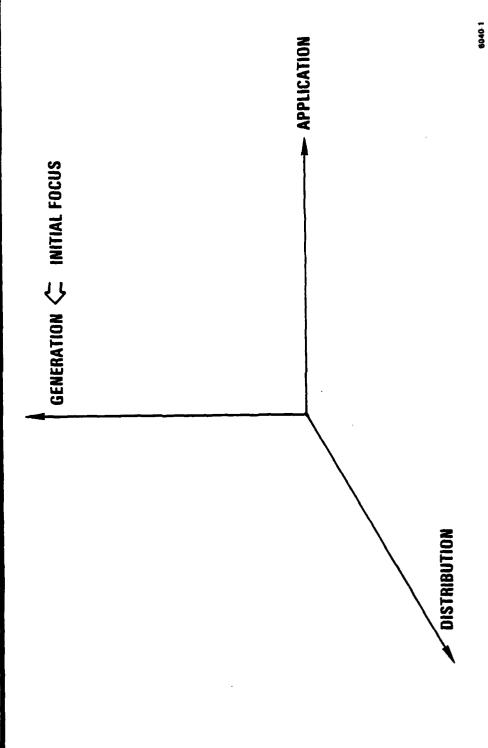
A. GARGARO F. PAPPAS

 $\mathsf{Ada}^{ extsf{B}}$ is a registered trademark of the U.S. Government Ada joint program office.

5040

REUSABILITY TECHNOLOGY FOCI CSC

Ċ,



- REUSABLE CODE GENERATION IS NOT A WELL-DEFINED DISCIPLINE CONSIDERING DIFFERENT
- □ EXECUTION ENVIRONMENTS
- LANGUAGES
- a APPLICATION DOMAINS
- □ PERFORMANCE REQUIREMENTS
- REUSABILITY NOT AN EXPLICIT REQUIREMENT OF MANY MCCR APPLICATIONS, **LOWERING POTENTIAL FOR GENERATING REUSABLE CODE UNITS**
- MANY CLAIMS REGARDING Ada LANGUAGE AND REUSABILITY ARE UNPROVEN

OBJECTIVE

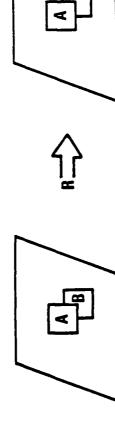
- UNDERSTAND ISSUES OF GENERATING POTENTIALLY REUSABLE CODE UNITS IN TRANSITIONING MCCR APPLICATIONS TO THE Ada LANGUAGE, EMPHASIZING:
- PUNDAMENTAL CRITERIA FOR EVALUATING REUSABILITY
- □ PERFORMANCE EFFICIENCY
- □ APPLICATION DEVELOPMENT PRACTICES

APPROACH

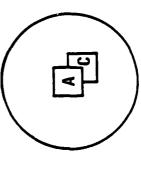
- LIMIT SCOPE TO RESTRICTED MCCR (REAL-TIME) DOMAIN
- REFINE EXISTING WORK TO RECOGNIZE:
- a THE USE OF CHAPTER 13
- **EXISTING TECHNIQUES WILL PERSEVERE**
- CREATE PEDAGOGICAL EXAMPLES TO ILLUSTRATE 130W TO "BUILD AND FIT" **STACHOUR ET AL] REUSABLE CODE UNITS TO MCCR APPLICATIONS**
- **DEVELOP A REUSABILITY HANDBOOK FOR MCCR APPLICATION DEVELOPERS TO ASSIST IN THE TRANSITION TO THE Ada LANGUAGE**



COMPONENT REUSABILITY











CODE PARAMENT RECORDOR DESCRIPTO L'ASSESSE PLASSESSE RECORDE RECORDE PROPERTIE D'ADMINISTRE PROPERTIE PROP

(<u>•</u>

REUSABILITY

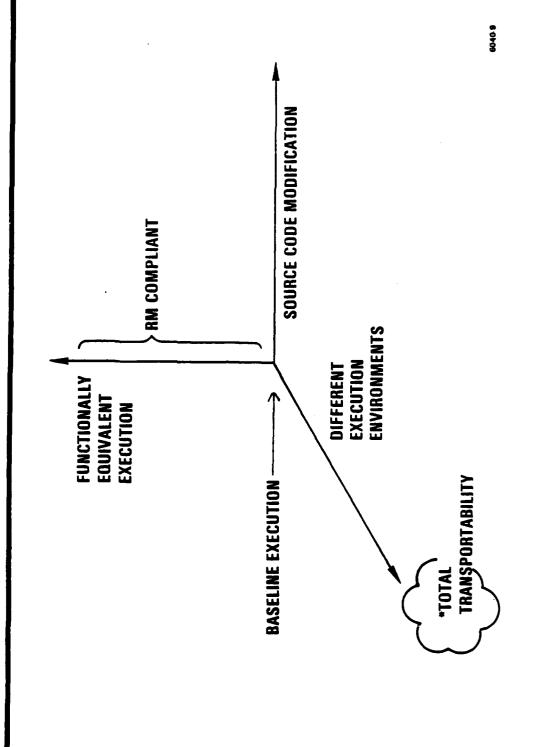
- REUSABILITY CRITERIA SPECIFY:
- DEST HARNESS, OR "SCAFFOLDING" [BENTLEY]
- □ FUNCTIONAL INDEPENDENCE OF PROGRAM'S CONSTITUENT PARTS TERMED COMPOSITION ORTHOGONALITY
- □ INDEPENDENCE OF THE IMPLEMENTATION OF THE "Ada RUN-TIME ENVIRONMENT" [KAMRAD et al]

- TRANSPORTABILITY CRITERIA INCLUDE:
- **EXECUTION OF COMPLETE PROGRAM**
- FUNCTIONALLY EQUIVALENT EXECUTION IN COMPLIANCE WITH RM – DOES NOT PRECLUDE GENERATING CODE THAT CAN BE **OPTIMIZED OR IS SENSITIVE TO EXECUTION ENVIRONMENT**

8

DEGREES OF TRANSPORTABILITY (_{MI}T CSC

8



CSC COMPOSITION ORTHOGONALITY

- COMPOSITION ORTHOGONALITY CONVEYS REUSABILITY DESIDERATA FOR **CONSTRUCTING PROGRAMS OR GENERATING CODE UNITS:**
- a TOP-DOWN VIEW PROMOTES UNDERSTANDING OF PROGRAM CONSTRUCTION THAT EMPHASIZES DEFENSIVE REUSABILITY, i.e., AVOIDING POTENTIAL IMPEDIMENTS TO REUSING CONSTITUENT CODE UNITS
- BOTTOM-UP VIEW PROMOTES UNDERSTANDING OF CODE UNIT GENERATION THAT EMPHASIZES AGGRESSIVE REUSABILITY, i.e., FORMAL GENERATION OF REUSABLE CODE UNITS
- ACHIEVABLE LEVELS OF REUSABILITY MAY BE IDENTIFIED AS: WEAK, **EFFECTIVE, AND STRONG**





THE PRODUCTION OF THE PRODUCT OF THE

CSC

WEAK REUSABILITY

```
Table: array (1.. Max_Table_Elements) of Element_Type;
```

```
function Binary_Search (Element : in Element_Type) return Natural is
```

Left_Point : Positive := 1;

Right_Point : Positive := Max_Table_Elements;

Mid_Point : Positive;

begin

```
while Left_Point <= Right_Point loop

Mid_Point := (Left_Point + Right_Point) / 2;

if Element < Table (Mid_Point) then

Right_Point := Mid_Point - 1;

elsif Table (Mid_Point) < Element then

Left_Point := Mid_Point + 1;

else

return Mid_Point;

end if;

end if;

return 0;
```

end Binary Search;

CSC STR

STRONG REUSABILITY — BODY

```
if Left End = Right End and then Element = Table (Left End) then
                                                                                                                                                                                                                                                                                               + Index_Type'Pos (Right_End) / 2);
function Binary_Search (Table: Table_Type; Element: Element_Type)
                                                                                                                                                                                                                                          while Left_End < Right_End loop
Mid_Point := Index_Type'Val (Index_Type'Pos (Left_End)
                                                                                                                                                                                                                                                                                                                        if Element < Table (Mid_Point) then
Right_End := Index_Type'Pred (Mid_Point);
                                                                                                                                                                                                                                                                                                                                                                            elsif Table (Mid_Point) < Element then Left_End := Index_Type'Succ (Mid_Point);
                      : Index_Type := Table'First;
                                                                              Right_End : Index_Type := Table Last;
                                                                                                                                                            if Table'Last < Table'First then
                                                                                                      Mid_Point : Index_Type;
                                                                                                                                                                                                                                                                                                                                                                                                                                                              return Mid_Point;
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            raise Not Found;
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      return Left_End;
                                                                                                                                                                                       raise Not_Found;
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             end Binary_Search;
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           end if:
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        end loop;
                                                       Left_End
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      end if;
```

6040-13

STRONG REUSABILITY — SPECIFICATION CSC

generic

```
type Index_Type is private;

type Index_Type is array (Index_Type range <>>) of Element_Type;

type Table_Type is array (Index_Type range <>>) of Element_Type;

with function "<" (Left, Right: Element_Type) return Boolean is <>;
```

package Binary Search Package Template is

function Binary_Search (Table: Table_Type; Element: Element_Type)
return Index_Type;

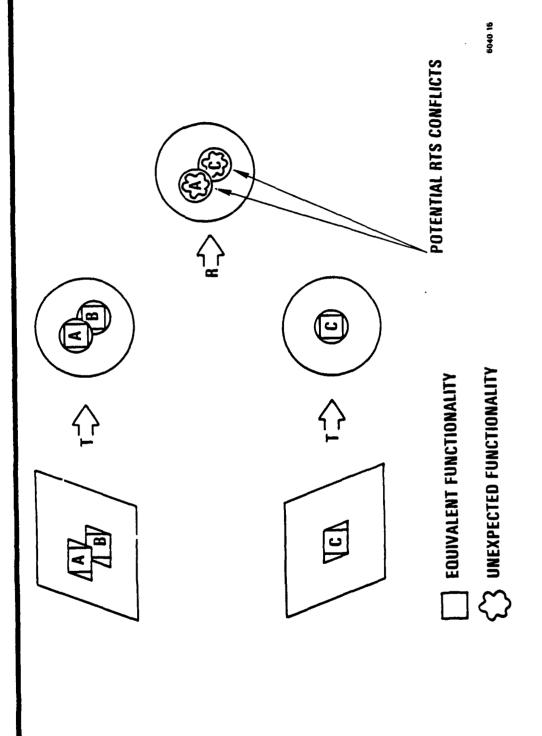
Not_Found: exception;

end Binary_Search_Package_Template;

INDEPENDENCE OF THE Ada RUN-TIME SYSTEM — (RTS)

- DEPENDENCIES UPON THE Ada RTS THAT COMPROMISE REUSABILITY MAY BE INTRODUCED, EXPLICITLY OR IMPLICITLY
- INDEPENDENCE OFTEN HIDDEN IN MORE GLOBAL REUSABILITY CHARACTERISTICS
- MCCR APPLICATIONS PARTICULARLY VULNERABLE
- SENSITIVITY TO PROBLEMS BECAUSE OF CONFLICTING RTS DEPENDENCIES COMBINING CODE UNITS DEVELOPED UNDER DIFFERENT RTS, INCREASES

CSC COMPONENT REUSABILITY AND RTS CONFLICTS



IMPLICIT RTS DEPENDENCE

```
Active_Readers := Active_Readers - 1;
                                                                                                                                                                       -- >>> Implicit dependency on stability of COUNT
-- Task controlling read/write access to shared variable
                                                                                                                          if Active_Readers = 0 then while Start_Write Count > 0 loop
                                                                                                                                                                                           accept Start_Write;
                                                                                                                                                                                                             accept Stop_Write;
                                                                                   accept Stop_Read;
                                                                                                                                                                                                                                    end loop;
                                                                                                                                                                                                                                                                                                 terminate;
                                                                                                                                                                                                                                                                                                                         end select;
                      select
                                                                5
```

-- procedure called by writer tasks

select

Rw_Control . Start_Write;

-- Update shared variable with actual parameter from call

delay Write_Time_Limit; Rw_Control. Start_Write;

-- Update shared variable to indicate that the writer was late

end select;

8040·1

CONCLUSIONS

- MCCR APPLICATIONS TO THE Ada LANGUAGE MAY HAVE BEEN OVERESTIMATED ■ THE POTENTIAL FOR GENERATING REUSABLE CODE UNITS IN TRANSITIONING
- SPECIFIC RECOMMENDATIONS TO INCREASE POTENTIAL FOR REUSABLE CODE **UNITS ARE:**
- DEFENSIVE REUSABILITY SHOULD BE INTRODUCED AT THE DESIGN LEVEL
- SPECIFICATION OF CLASSES OF Ada VIRTUAL MACHINES SHOULD INCREASE **POTENTIAL FOR REUSABILITY**
- FORMAL SPECIFICATION OF REUSABLE CODE UNITS SHOULD BE RESEARCHED **AS A MEANS FOR IDENTIFYING REUSABILITY VIOLATIONS**

COSMIC - NASA's Software Distribution Center

John A Gibson

COSMIC Computer Services Annex University of Georgia Athens, GA 30602

Abstract

NASA and the University of Georgia established the Computer Software Management and Information Center (COSMIC) to collect and disseminate computer software developed by NASA and its contractors. The current COSMIC inventory of over 1100 programs is available for business, industry, educational institutions, and government.

Text

The Computer Software Management and Information Center (COSMIC) was established by NASA and the University of Georgia in 1966 to function as a software collection center and to provide dissemination service for computer software developed by NASA and its contractors. COSMIC has received and processed nearly 5000 computer programs since its beginning. Currently over 1100 programs in the inventory are supplied to business, industry and educational institutions as well as to other government agencies. Programs are priced at a fraction of their original development costs.

Each new computer program and document and each update received from NASA and NASA contractors is screened and evaluated for completeness and application potential before being added to the inventory. This process involves checking for syntactical accuracy through compilation and/or assembly on a host of systems available at the University of Georgia. Each program is assigned appropriate subject category codes and index terms before an abstract is prepared and the program is made available to the public.

The software submitted to COSMIC reflects the varied activities of NASA which involve basic research and development projects as well as projects directly related to space missions. Software developed in such areas as structural mechanics, computer

graphics, mathematics, communications, and thermodynamics broaden the scope of programs in the inventory. Many of these programs can be directly applied to secondary use with little or no modification. Other programs can be adapted for a very specific purpose at substantially less than the cost of developing a new program. COSMIC supplied the source code with each program so that its capabilities can be modified or extended as needed.

The COSMIC customer service staff provides assistance to users in locating programs or groups of programs that best meet the user's needs. This customized search by the COSMIC staff is provided at no charge to the user. The COSMIC staff is trained to assist users in locating software and will assist in locating specific public domain software packages even if they are not part of the COSMIC inventory. For users who have a general interest in software or for broad application needs, COSMIC publishes the COSMIC Software Catalog. This annual publication is a comprehensive collection of program abstracts, organized into 75 subject categories and includes a keyword index and an author index to aid in the location of programs.

Our users provide the best examples of how NASA software is used. These examples include: 1) using the application package essentially as-developed for a similar application in industry, 2) converting the application



package to operate on a different machine, 3) and taking related routines from one package and applying these routines in a different package. COSMIC's service includes distribution of programs and documents between NASA centers, so our users include many NASA staff members. Approximately 25% of COSMIC's distribution involves the transfer of software to NASA centers and contractors for reuse on NASA projects.

In 20 years of experience operating NASA's software distribution COSMIC has had many opportunities to learn. The lessons we have learned cover many of the items mentioned in the workshop announcement letter under "library experience" and "logistics of reuse". The best advice I can give your library committee is to keep the number of rules, directives, restrictions and paper work to a minimum. Make it easy to put programs into your library. Make sure that your staff will be friendly and helpful in locating software for a user. Make sure you have an efficient system for transmitting the software to the user. Do all your screening, testing, quality assurance, performance measurements, etc., before the software officially becomes available from the library. Define the technical or user support available for each item in the inventory. Last, but not least, obtain information from users that reflect the benefits they realized from using the software.

The actual utilization of the library as a place to submit software as well as a place to obtain software will depend on your ability to market your services. Our experience shows that this effort, both to obtain software and to promote the use of software, is a continuing effort that involves significant resources of staff time and money.

The concept of a single source of computer software, whether routine or application packages, is not new. NASA has 20 years experience in operating such a facility, COSMIC, at the University of Georgia.

COSMIC®

NASA'S COMPUTER SOFTWARE MANAGEMENT

AND

INFORMATION CENTER



"The aeronautical and space activities of the United States shall be conducted so as to contribute . . . to the expansion of human knowledge of phenomena in the atmosphere and space. The Administration shall provide for the widest practicable and appropriate dissemination of information concerning its activities and the results thereof."

-NATIONAL AERONAUTICS AND SPACE ACT OF 1958



 $\mathcal{R}_{\mathcal{P}}$

COSMIC ACTIVITIES

- 1. Technical Screening
- 2. Promotions
- 3. Order Processing
- 4. User Support
- 5. NASTRAN Maintenance
- 6. Benefits Analysis

HARDWARE AVAILABLE FOR PROGRAM CHECKOUT

CDC CYBER 205

CDC CYBER 845

CDC CYBER 850

IBM 3081

DEC VAX 11/780

UNIVAC 90/80

MICROCOMPUTERS

DOCUMENTATION REQUIREMENTS

Problem / Function Definition

Method of Solution

User Instructions

Implementation Instructions

Sample Input / Output

Environmental Characteristics

Other Appropriate Information



- Excellent quality program. Qualifies as Tech Brief. Must be NASA funded.
- Program and documentation meet publication standards.
- III Programs returned to the submittal site.
- IV Programs and documentation which are incomplete and additional information has been requested.

PRICING FACTORS

Machine independence and / or vintage

Level of programming or maintenance support

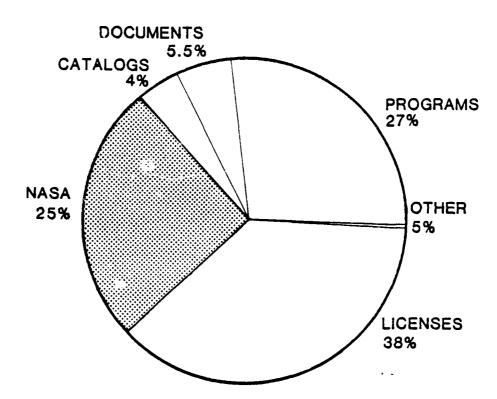
Quality of supporting documentation

Program sales potential or history

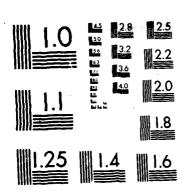
Program functionality

Program size

COSMIC PERCENTAGE DISSEMINATION BASED ON DOLLAR VALUE 1984



SOFTMARE TECHNOLOGY FOR ADAPTABLE RELIABLE SYSTEMS (STARS) MORKSHOP MARCH 24-27 1986(U) NAVAL RESEARCH LAB WASHINGTON DC MAR 86 AD-A198 128 UNCLASSIFIED F/G 12/5 NL



G MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

THE DESIGN FOR REUSABLE SOFTWARE COMMONALITY

Norman S. Nise Chuck Griffin

Rockwell International

Abstract

This paper discusses the design of software packages to improve their degree of reusability. The degree of commonality both across applications areas and within applications areas are tied together to form a commonality matrix against which software packages can be measured for potential reuse. Further, design techniques to improve commonality are explored. Emphasis is placed upon designing software with the widest domain of applicability in order to realize the financial benefits of reusable software.

Introduction

By now it is a well known fact that the demand for software is increasing disproportionately to the supply. More specifically, the demand is increasing at a 12% rate compared to an increase of only 8% for software supply. The problem is exacerbated by the fact that software costs are also rising - astronomically. The United States Department of Defense estimates that software costs will rise to \$32 billion per year from a modest \$2.5 billion in 1980.

Software development methods have contributed to the software economic problem through maintenance costs that represent 50 to 80 percent of the total cost while design and development are only 20 to 50 percent of the cost. These methods have also resulted in code that is difficult to modify, contains errors, and is produced with low productivity.

The DOD and its Software Technology for Adaptable Reliable Systems (STARS) initiative have already begun to study ways to alleviate the above described problems.

One promising solution to the software problem is reusable software. Reusable software includes reusable requirements, specifications, design, code, documentation, etc. It can be used between applications with little or no modification. It can be imported

* Ada is a registered trademark of the United States DOD, Ada Joint Program Office.

and used as part of a larger project. Reusable software will cause a reduction in manhours required for the design, development, testing and maintenance of software.

Two major factors have prevented the idea of reusable software from becoming a reality. The first was the proliferation of computer languages. It would have been a difficult task to keep and catalogue software over the domain of many languages. With the development of Ada* along with the DOD's decree that Ada will be the higher level language for all embedded systems, the outlook for the future is a single, widely-used language that can be used to develop and use reusable packages. Ada itself was designed to be used for the development of reusable packages.

The second factor has been proprietary interests on the part of software developers. There was the fear of not realizing maximum profits if software was shared among other members of the industrial community. Developed software was hidden and not shared. STARS is now looking into this problem by trying to establish incentives to develop and use reusable software.

We can envision, some time in the future, a reusable software library where software "parts" are cataloged and available for use within larger programs.

Commonality

One of the factors that will determine the success of a library package will be the degree of reuse. The more times a software package is used, the more we can rely on a reusable software system to solve the economic problems previously described. The degree of reuse depends upon the domain of applicability of that software. That is to say, can that software package be depended upon to be used many times? If the software has wide applicability either across many different applications or wide applicability within a single application, the answer probably will be affirmative.

The amount of reuse to which a module is put through is dependent upon letting the user know of its existence and the domain of its applicability. Thus, it will be important for the designer of the package to give to that package the attribute of maximum applicability and properly classify that package as to its range of application. Again, the designer must not only build the attribute of wide applicability into the package, but also must communicate this attribute to the user of the library.

It is obvious that several pitfalls can be encountered that will diminish the economic benefit to be accrued from the use of reusable software.

- (1) wide applicability is built into the A package but that attribute is not communicated to the user through the library classification system,
- (2) a package is designed that has wide applicability over a narrow field of applications but could have been designed to cover other application areas,
- (3) a package is designed that has narrow applicability but could have been designed to have wider applicability either over a single applications area or over many applications areas.

Thus proper design and classification up front is imperative. Narrowing of the field of applicability will lead to the proliferation of modules with the resulting increase in cost along with the unneccessary complication of the reusable software retrieval system. More

software will exist and maintenance costs will increase.

If a software package is classified as application specific, the likelihood of the package being applied outside of that domain will be small. For example, software classified as being in the domain of accounting, will be used only for accounting. That package will not be used for missile systems. If the package contained sort routines that could be used outside of the domain of accounting, the savings would not be realized in this case.

As reusable software libraries are established it is imperative that software placed into these libraries be designed with as large a domain of applicability as possible.

For example, a routine to add two objects together could be very specific and be applicable only to integer numbers. Other packages would then have to developed for floating point numbers, fixed point numbers and the like. Each package will have diminished reuse and each package will multiply the costs of development and maintenance. We cannot overemphasize the fact that library package development use every technique possible to extract, up front, from a package the maximum amount of reuse. To assume application specificity when in reality wide domain applications can be obtained is to defeat the gain to be realized from reusable software.

Let us first take a look at a classification system to describe the commonality of software parts.

Classification System for Commonality

Application software reuse can be measured across two domains:

- (1) within an application area
- (2) across application areas

By applications area we mean a distinct industrial grouping. For example, different applications areas could include missiles, aircraft, spacecraft, weapons, ships, lasers, command/control, radar, etc.

Software that fulfills a high degree of reuse in any of the above domains is a good



candidate for reusable software. Since our objective is to design software packages that will yield the maximum amount of reuse, the design process should explore the possibility of expanding a package designed for a specific applications are to a package that is reused across many applications areas. Again, the cost of not looking carefully into this possibility and filling a reusable software library with an excess of application specific software packages could cancel some of the benefits that would accrue to a reusable software system.

We can then think of commonality matrix that has two dimensions:

- (1) degree of commonality across application areas
- (2) degree of commonality within an applications area

This matrix is shown in Figure 1. Each square of the matrix suggests a relative value for software package commonality. The higher the index, the greater the domain of applicability that is predicted. The scale goes from 0 to 6 with 0 yielding the least commonality and 6 yielding the most.

As an example, Figure 2 suggests a possible classification of software for a spacecraft application specific area. This classification then classifies the given software according to the vertical direction of Figure 1. In Figure 3 the same software packages are classified according to the horizontal direction of Figure 1. If we locate the intersection of each package on Figure 1, we can determine the commonality index. We now list each package and its commonality index:

sorts	6
data structures	.6
abstract processes	6
computer system	6
s/w maintenance	6
math functions	5
geometric functions	5
matrix functions	5
vector functions	5
process functions	5
communications	5
guidance functions	4
navigation	
telemetric functions.	4

s/w design	4
s/w development	4
s/w verification	4
mission functions	2
input routines	2
output routines	
system functions	
warhead control	
system inputs	
system output	

This matrix can be used then to orient us in the design of reusable packages. Our design objective is to improve the packages' commonality index. Let us now take a look at techniques that can help us meet our goal of increasing a packages' commonality index.

Reusable software will be designed using Ada with its attributes of information hiding, modularity, and generic packages. In order to improve the commonality index of an Ada software package, we want to strip away those parts of the package that contribute to a narrow degree of applicability. Figure 4 shows a package divided into its application specific parts and its application independent parts. Here we are dividing the package into three main divisions; (1) input, (2) process, (3) output.

We can assume that the package of Figure 4 would not be a good candidate for reusable software across application boundaries because of the application dependent parts. If the application dependent parts are removed, the remainder of the package could possibly be engaged in heavy reuse.

Two ways exist to solve this problem. The first way would be to create application dependent packages consisting solely of the application dependent parts of the original package. This concept is shown in Figure 5. This technique would create two library packages. One package would have a high degree of commonality and reuse while the other package would have a low degree of commonality and reuse.

A second technique would be creation of generic packages that would be instantiated with the application dependent parts as shown in Figure 6. Here the only library package is

the generic packages. The generic packages would have the most reuse across applications boundaries. The instantiator is not a library package, but rather is a software module created for an application specific function. Its reuse would not be great and it would not be placed into the reusable software library.

Figure 6 shows the input, process, and output within the same package. It might be preferable to keep the input, process, and output in separate and distinct packages. For this case, the instantiator procedure would require sequencing in order to instantiate the input, process, and output packages in the proper order.

Example

As an example, consider the software represented in Figure 7. This software checks inputs for limits, ranges, and, in the case of discrete inputs, desirable states. The outputs from the software are various messages along with scaled inputs.

This module thus contains much in the way of application specific software and tables. It contains limits, scale factors, messages, and the like. This package would not be considered to be reusable. Why, every application would require a reconfiguration.

In order to make this package reusable, the non-generic parts can be removed. A generic module that does the checking, scaling and data output can be written. The reusable module would contain just the process. The data can be acquired from other modules that are application specific.

Figure 8 shows the generic module used for checking, scaling, and the outputting of messages and scaled data. Other application specific modules handle conversion to common data types and contain tables of ranges, limits, scale factors, and messages along with tables formed from the converted inputs. In Figure 8, modules 1 and 2 would not be reusable, but module 3 would. Modules 1 and 2 could be contained within the instantiator. Module 3 now can be used over a wide range of applications where range and limit checking are done.

This example shows that with proper design, reusable modules can be created from non-reusable modules by separating the application specific parts and creating generic reusable modules.

Summary

This paper has presented several concepts relating to the design of software packages with the attribute of high reusability. Specifically, we showed that the degree of commonality of a module must be measured both within a specific applications area and across many application areas. A matrix was presented with which to evaluate the commonality of modules across both domains.

Two techniques were presented for improving the commonality of software packages. The first technique suggests separating application dependent parts from application independent parts. The application independent part becomes a reusable package.

The second technique creates a generic package that is instantiated with application dependent parameters. The generic package becomes the reusable software package while the instantiator is developed for each application and is not a reusable software package.

Acknowledgements

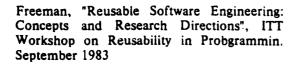
The authors wish to thank Keith Morris for his invaluable input in the preparation of this paper.

McCain, "A Software Development Methodology for Reusable Components", STARS Workshop 1985 Reports.

McNicholl & Anderson, "CAMP Preliminary Technical Report", STARS Workshop 1985 Reports

Snodgrass, "Fundamental Technical Issues of Reusing Mission Critical Application Software", STARS Workshop 1985 Reports

Common Ada Missile Packages, Interim Report AFATL-TR-85-17, September 1984 -January 1985



Nise, Dillehunt, McKay, Kim, Griffin, "A Reusable Software System", AIAA/ACM/NASA/IEEE Computers in Aerospace V Conference, 21-23 October 1985

Grabow & Noble, "Reusable Software Concepts and Software Development Methodologies", AIAA/ACM/NASA/IEEE Computers in Aerospace V Conference, 21-23 October 1985

McCain, "Reusable Software Component Construction, A Product-Oriented Paradigm", AIAA/ACM/NASA/IEEE Computers in Aerospace V Conference, 21-23 October 1985

Jones etal, "Issues in Software Reusability", SigAda

Parnas, "On the Criteria to be Used in Decomposing Systems into Modules", Communications of ACM, 1972

Parnas, "Designing Software for Ease of Extension and Contraction", IEEE Transactions on Software Engineering, March 1979

Horowitz & Munson, "An Expansive View of Reusable Software", ITT Workshop on Reusability in Programming, September 1983

Goodell, "Quantitative Study of Functional Commonality in a Sample of Business Applications", ITT Workshop on Reusability in Programming, September 1983

Figure 1 - Commonality Matrix

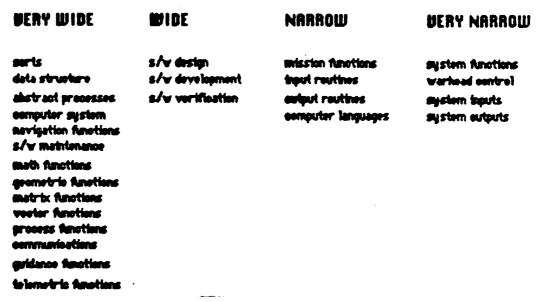


Figure 2 - Classification of Commonality. Example Spacecraft Specific.

BERY WIDE

serts
data structure
abstract processes
computer system
computer languages
s/w maritenance

WIDE

math functions
geometric functions
matrix functions
vector functions
process functions
communications
s/w design
s/w development
s/w verification

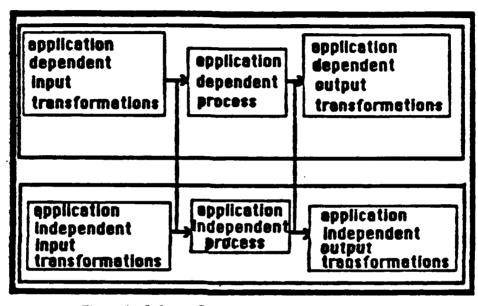
NARROW

navigation functions guidance functions mission functions telemetric functions input routines eutput routines

BERY WARROW

system functions warhead control system seputs system cutputs

Figure 3 - Classification of Commonality. Example Across Applications.



ETAJI

Figure 4 - Software Package Containing Application Parts

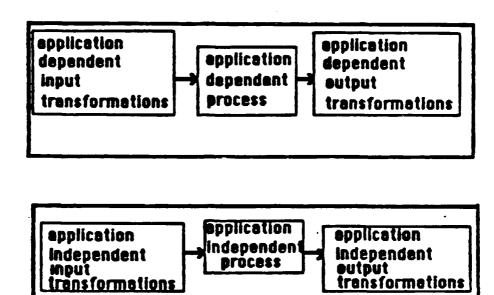


Figure 5 - Packages Separated Into Application Independent and Application Dependent

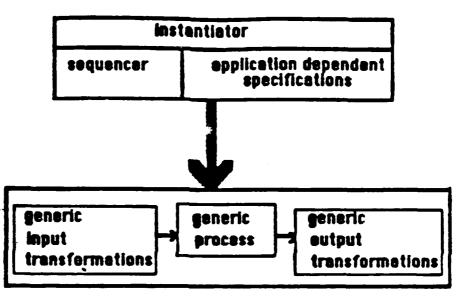


Figure 6 - Figure Application Dependent Parts Created from Instantiation of Generic Packages

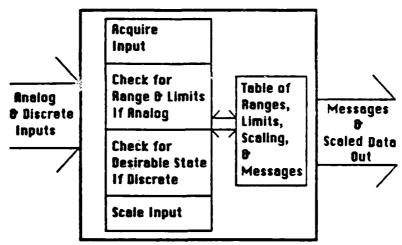


Figure 7 - Non-Reusable Scaler-Checker

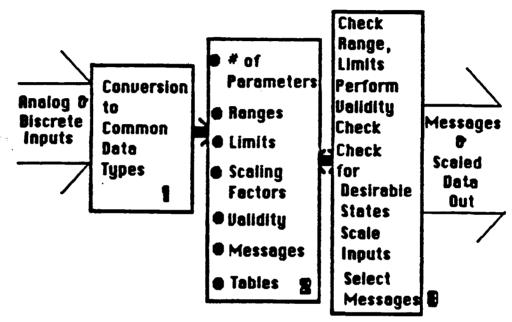


Figure 8 - Separate Reusable (Block 3) and Non-Reusable (Blocks 1 and 2) Packages

(

800

THE DESIGN FOR REUSABLE SOFTWARE COMMONALITY

by

Norman S. Nise Chuck Giffin

Rockwell International Downey, California



THE DESIGN FOR REUSABLE SOFTWARE COMMONALITY

COMMONALITY

CLASSIFICATION SYSTEM FOR COMMONALITY

PACKAGE DESIGN TO IMPROVE COMMONALITY

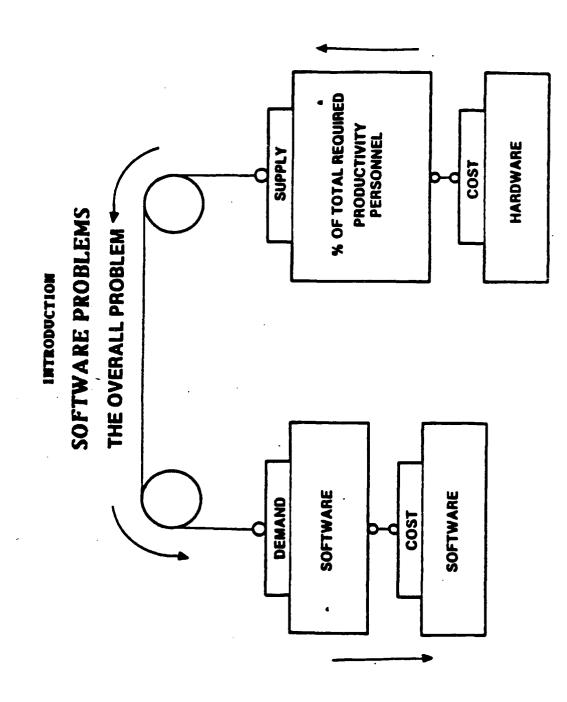
AN EXAMPLE

INTRODUCTION

PURPOSE OF THE BRIEFING

- TO STATE THE IMPORTANCE OF DESIGNING ADA REUSABLE PACKAGES, APRIORI, WITH A DOMAIN OF MAXIMUM APPLICABILITY
- TO MAKE SUGGESTIONS TO ACCOMPLISH THE TASK

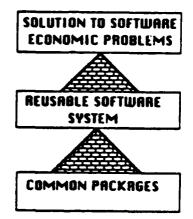
Ada is a registered trademark of the United States DoD, Ada Joint Program Office



W)

INTRODUCTION

THE IMPORTANCE OF SOFTWARE COMMONALITY



THE DESIGN FOR REUSABLE SOFTWARE COMMONALITY

INTRODUCTION

COMMONALITY

CLASSIFICATION SYSTEM FOR COMMONALITY

PACKAGE DESIGN TO IMPROVE COMMONALITY

AN EXAMPLE

THE COMMONALITY ISSUE

- 1. LIMITED DOMAIN OF APPLICABILITY
 - Package designed for no reuse within an applications area
 - Package designed for reuse within an applications area

INSTEAD OF

Designed for reuse across applications areas

2. DOMAIN OF APPLICABILITY IS NOT COMMUNICATED TO THE SOFTWARE DESIGNER

WHY THERE IS A COMMONALITY ISSUE

• Difficulty in establishing guidelines for reuse across applications

applications vs functions

• Lack of incentives for developers in a single applications area

SANCE SANCE

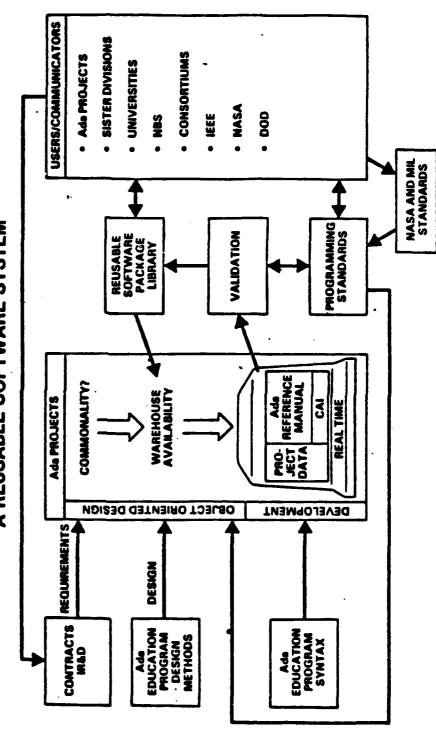
CONTRACT PROPERTY OF STREET, S

IMPLICATIONS OF THE COMMONALITY ISSUE

- Packages with little reuse
- Proliferation of similar packages within the reusable software library
- Difficulty in retrieval and classification
- More packages mean more development and maintenance costs
 - Possible cancelation of the economic benefits of a reusable software system

SOLUTIONS

A REUSABLE SOFTWARE SYSTEM



COMMONALITY

APPLICATIONS VS FUNCTIONS

	SORTS					
	FILTERS SORTS					
	- NO		1)
	GUID- FLIGHT ON -	•	•	1)
	GUID-	•	•			
	NAVI - Gation	•	•			
FUNCTIONS	APPLICATIONS	SPACECRAFT	AIRCRAFT	MISSILES	SATELLITES	

THE DESIGN FOR REUSABLE SOFTWARE COMMONALITY

INTRODUCTION COMMONALITY

PACKAGE DESIGN TO IMPROVE COMMONALITY
AN EXAMPLE

COMMONALITY MATRIX

WITHIN APPLICATIONS APPLICATIONS	VERT NARROW	NARROW	WIDE	VERY WIDE
VERT NARROW	0	1	2	3
NARROW	1	2	3	4
WIDE .	2	3	4	5
VERY WIDE	3	4	5	6

CLASS IFICATION SYSTEM FOR COMMONALITY

137

EXAMPLE - SPACECRAFT SPECIFIC

system functions warbend control

VERT HARROW

system inputs system outputs

VERY WIDE	Wide	MARROW
sorts	s/w design	mission functions
data structures	s/w development	input routines
abstract processes	/w verification	output routines
computer system	•	computer languages
navigation functions		
s/w maintenance		
math functions		
geometric functions		
matrix functions		
vector functions		
process functions		,
communications		
guidance functions		
telemetric functions		

CLASS IFICATION SYSTEM FOR COMMONALITY

EXAMPLE - ACROSS APPLICATIONS

VERT WIDE	orts	ndstract processes	computer system	computer languages	6/w maintenance
VER	sorts	abst	COB	E 00	3/5

WIDE math functions geometric functions geometric functions matrix functions vector functions process functions process functions s/w design

s/w development

s/w verification

CLASSIFICATION SYSTEM FOR COMMONALITY

EXAMPLE - COMMONALITY RATING

geometric functions 5 system function 5 warhead control 5 vector functions 5 system inputs 5 process functions 5 system outputs 5 communications 5 s/w design 4 s/w development 4 s/w verification 4 computer languages 4	s (ons (o)	22000
---	------------	-------

THE DESIGN FOR REUSABLE SOFTWARE COMMONALITY

INTRODUCTION

COMMONALITY

CLASSIFICATION SYSTEM FOR COMMONALITY

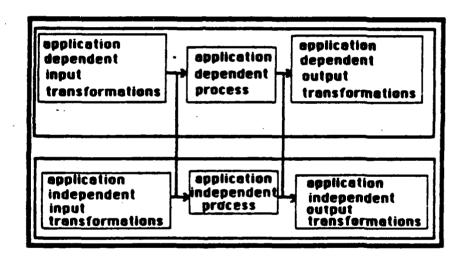
PACKAGE DESIGN TO IMPROVE COMMONALITY

AN EXAMPLE

PACKAGE DESIGN TO IMPROVE COMMONALITY

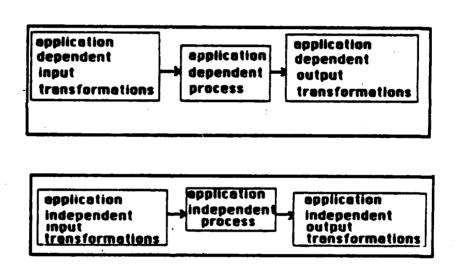
88

SOFTWARE PACKAGE CONTAINING APPLICATION DEPENDENT PARTS



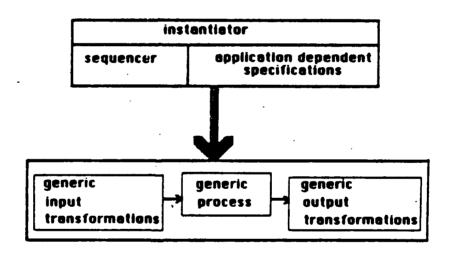
PACKAGE DESIGN TO IMPROVE COMMONALITY

PACKAGES SEPARATED INTO APPLICATION INDEPENDENT AND APPLICATION DEPENDENT PARTS



PACKAGE DESIGN TO IMPROVE COMMONALITY

APPLICATION DEPENDENT PARTS CREATED FROM INSTANTIATION OF GENERIC PACKAGES



THE DESIGN FOR REUSABLE SOFTWARE COMMONALITY

INTRODUCTION

COMMONALITY

CLASSIFICATION SYSTEM FOR COMMONALITY

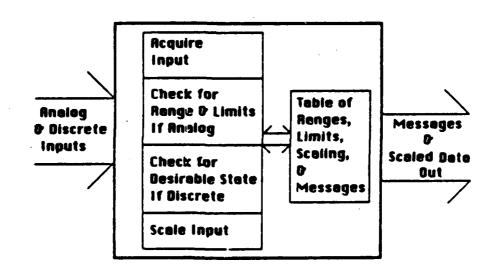
PACKAGE DESIGN TO IMPROVE COMMONALITY

AN EXAMPLE



AN EXAMPLE

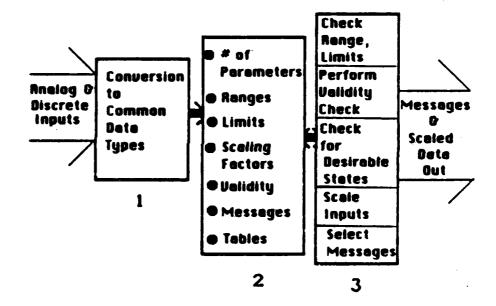
NON-REUSABLE SCALER-CHECKER





AN EXAMPLE

SEPARATE REUSABLE (BLOCK 3) AND NON-REUSABLE (BLOCKS 1 & 2) PACKAGES FOR SCALER-CHECKER



MSAT Brief Narrative to Accompany the High Level Technical Brief

C. Ogden

Slide 1: The Software Environment

Software System Size/Complexity - many of the current Army systems contain software with hundreds of functions are hundreds of thousands of lines of source code. To manually analyze this volume of software would be extremely labor intensive.

Software Costs - Software often takes the lion share of the system cost because software development is so labor intensive. Software maintenance can account for as much as 75% of the Life cycle cost.

Proliferation of languages/dialects - It is estimated that there are 450 software languages in existence; even though Ada is suppose to be the cure-all and be the language for embedded systems applications, our experience has shown that a sizable percentage of the 450 languages are currently used by DoD in the systems we test.

Compliance with design methodologies/ development standards - We are all concerned that the software being developed for our applications complies with good design methodologies (such as top down design and structured programming) and good development standards (such as DoD-STD-2167). we check and verify compliance?

Slide 2: Systems Requiring Assessment

Back in April 1984, we took a look at the systems on which we were performing software testing or planning for software testing. This slide shows whose systems and the languages the software was written in.

Slide 3: Language Processors Required

This slide shows the languages of the previous slide ordered by High Order Language (HOL) and Assembler (ASM). To prepare automated tools to analyze software written in each of these languages would be very expensive. To manually analyze the software requires individuals familiar with

each of these languages and would also be very labor intensive. The number of languages with which we were bombarded, led us to develop an automated tool which, with a relative small effort, could analyze software written in a new language.

Slide 4: MSAT History

This slide shows the tools which led up to the MSAT effort. It started with the Program Flow Analyzer (PFA) back in the late 70's and early 80's. During 83 and 84, we developed specific tools for specific systems; but since they were written for specific systems and contained little documentations they are not reusable. During 84 and 85, we developed a FORTRAN Code Analysis Program (FCAP), a table driven Assembler Code Analysis Program (ACAP) and a C Code Analysis Program.

These are better documented and somewhat more general purpose in nature. However, they were just stop-gap measures in preparation of the general purpose tool - MSAT - which we are finishing the acceptance testing in April 86. The languages shown are languages which we have analyzed with one of the tools. The systems shown are systems which have had some of their software analyzed with one of the tools (not all results from these analyses have been included in formal test reports or the respective systems.

Slide 5: Development Philosophy

In developing MSAT our development philosophy included the following items: We wanted MSAT to be language table driven. We chose to use a Commercial Data Base Management System (DBMS) (INGRES) in order to minimize development costs and to take advantage of the many useful features of a commercial DBMS. We wanted MSAT to be user friendly, e.g., menu driven with lots of help. We knew that MSAT was not initially going to be the ultimate tool, so we

designed it for expansion and enhancement. We decided that we should practice what we preach so we fully documented MSAT (we followed DoD-STD-2167, draft standard at the time we started) and we even plan on running MSAT on itself to prove its quality and maintainability. We are validating the tool and the test plans, procedures and results are being reviewed by other Army organizations. We plan on maintaining configuration control of MSAT for many years.

Slide 6: Static Analysis Functions

This slide shows the 15 static analysis functions as delineated in the National Bureau of Standards (NBS) document: a taxonomy of tool features for the Ada programming support environment. The initial implementation of MSAT contains features in the following areas: Auditing, Complexity, Statistical Analysis, Interface Analysis, Comparison, Error checking and Structure Checking.

Auditing - comparing collected metrics to standards, e.g. from DoD-STD-1679 # Executable Statements = 200.

Complexity - McCabes Cyclomatic Complexity and Meyer's Extension

Statistical Analysis - Various simple statistics on the code metrics

Interface Analysis - Does the call statement with parameters match the called routine with its parameters

Comparison - Compares two version for structure changes, metric changes, etc.

Error Checking - various errors such as unresolved external references

Structure Checking - Recursion, Lower level module calling higher module

The Remainder of the functions should be added in the future.

Slide 7: MSAT Schematic

This schematic shows that MSAT runs on VAX with VMS. A tape of the source code (in ASCII) is entered onto disk. This source code is entered into the automatic Language processor which extracts the basic metrics and stores away the info into the data base. The Static analysis function take the

metrics from the data base and provides the various analyses and stores the results back into the data base. The report generator takes the info from the data base to create the desired reports.

Slide 8: MSAT Data/Control Flow

This slide provides a more detailed look of the data and control flow. It should be noted that the source code is entered into the automatic language processor along with the source Language description. For the static analysis the user can enter his own software standards to compare against the default standards in the data base which are based upon DoD-STD-2167, 1679.

Slide 9: MSAT Operational Capability

This slide shows the MSAT initial capability. The first language capability is VAX FORTRAN and 8085 Assembler. VAX FORTRAN was chosen because MSAT is written in VAX FORTRAN and we wanted to check the quality of the MSAT code itself. (MSAT also has the embedded query Language EQUEL for INGRES). MSAT is able to handle 3 languages for each system and 2 for any unit or subroutine. In a particular unit MSAT can analyze the code for a situation where there is embedded assembly language over the Static Analysis (SA) functions. The reports are described in greater detail on the next slide.

Slide 10: MSAT Reports

This slide shows the general break-down of the MSAT reports.

Source Listing/Table of Contents - We sometimes get boxes of code listings where a table of contents would come in very handy.

Software quality metric reports - This shows the various levels of the quality metric reports.

Structure Chart - This shows the subroutine hierarchical call structure

Error Report - Shows various errors detected i.e., Unresolved external references.

Interface Analysis Report - Potential problems in how the calling and called routines match up. Standards Compliance - Shows the various levels of this reports exception (those units not complying), unit summary, and system summary.

Change analysis report - the differences between two versions of the same system; shows differences in the metrics on a unit by unit basis as well as differences in system structure. This is useful to handle the tape-of-the-month syndrome where we test a system, we find problems, the contractor goes back to his place and comes back with a new version of the software to be installed. What did he change? This will help point where and what kind of changes have been made.

Slide 11: Sample report

This is a sample of MSAT Standards Compliance Unit Summary Report. It shows for example that the standard of executable Statements less than or equal to 200 is met by all 203 units. The next standard, Maximum consecutive lines of code without comments less than or equal to 10 is met by 192 or 94.6% of the units and not met by 11 units. The value 10 in the standard can be changed to 5 by the user if so desired.

Slide 12: Software Life Cycle Costs

Assume that a software system during the initial project development cost \$3M (as depicted by the solid line). Over the total life cycle of the system (10° years) the total software costs could total \$10M, with \$7M being spent for maintenance where maintenance is defined as fixing bugs and enhancing the system. Studies have shown that up to 75% of the software life-cycle costs can be spent on maintenance. Conversely, the dashed line depicts a system where we may have spent more money up front on documentation and using tools such as MSAT but the total life cycle costs should be decreased.

Slide 13: The Multi-Lingual Static Analysis Too! (MSAT)

This is an overview slide to refresh our memories as to what MSAT does. First, MSAT provides static analysis of the software in the areas we have already talked about. Second, MSAT provides a quality analysis of

the code, i.e., how good is the quality of the code. Third, it provides visibility into the code to assist or analysis in understanding and analyzing the code. Finally, MSAT provides a comparison between various version to show what has actually been changed and where.

Slide 14: Anticipated MSAT Users

Software Developers - we would like to give MSAT to software developer to be used during development - no reason to check the code after the development is complete, find problems and the developer says, OK there are problems, now pay me to fix them. Much better to give it to developer to be used during development so problems can be changed up front.

Verification and validation teams or contractors - obvious usage

Development testers - that us (EPG) can be used during DT.

Users - the Irtel School was interested in MSAT to help them analyze a program they had gotten from Great Britain to understand the source and algorithms.

Software Support Centers - the LCSSC could use MSAT for maintenance quality assurance.

Software Libraries - STARS is interested in MSAT to be used to analyze the reusable Ada components which will be placed in the reusable components library.

Slide 15: Future Development Proposed to STARS

The first thing is to develop the initial capability to analyze Ada code. Second, we need to make MSAT transportable so it can be more usable. To do this we propose rewriting MSAT in Ada (everybody should eventually have an Ada compiler), eliminating the VAX/VMS dependencies (system calls, etc.) and providing a stand-alone, government owned DBMS so that none has to buy the commercial DBMS.

Third, we need to enhance the Ada capability to handle the Ada special characteristics such as concurrent processing,

exception handling. Fourth, develop a library language capabilities: pick the 20 most used languages in DoD and make MSAT work on them.

Fifth, expand the current static analysis capability in all 15 areas shown before. Finally, provide the capability to store and report on manually collected data such as software trouble reports associated with the software system.

Slide 16: FY86 MSAS Goals

If MSAS is funded by STARS by April 1, the initial Ada capability would be developed, we would pick another language such as Pascal and provide the capability to analyze software written in that language. And finally, we would begin converting MSAS to Ada to make it transportable.

MULTILINGUAL STATIC ANALYSIS SYSTEM (MSAS)

INTRODUCTION

- A. HISTORY
- B. DEVELOPMENT PHILOSOPHY
- C. DESIGN FEATURES
- D. REPORTS
- E. BENEFITS
- F. ANTICIPATED USES
- G. PROPOSED DEVELOPMENT

THE SOFTWARE ENVIRONMENT

- SOFTWARE SYSTEM SIZE/COMPLEXITY
- SOFTWARE COSTS
- PROLIFERATION OF LANGUAGES/DIALECTS
- CUMPLIANCE WITH DESIGN METHODOLOGIES/DEVELOPMENT STANDARDS
 - TOP DOWN DESIGN
 - STRUCTURED PROGRAMMING
 - SOFTWARE DEVELOPMENT STANDARD DOD-STD-2167



SYSTEMS REQUIRING ASSESSMENT (April 1984)

SYSTEM UNDER TEST	LANGUAGE
Teampack	ROLM 1602 ASM
RPV	FORTRAN IV (DEC) PL/M-80 SKC FORTRAN 8085 ASM MACRO-11 ASM SKC 3121 ASM
JTIDS	SKC FORTRAN SKC 3132 ASM AMZ 8002 ASM
REGENCY NET	MICROTEK PASCAL OMSI PASCAL 8085 ASM AMD 2901 ASM RCA 1802 ASM
TRAILBLAZER	C ROLM FORTRAN 68000 ASM



LANGUAGE PROCESSORS REQUIRED

HOL ASM

C ROLM 1602

FORTRAN IV (DEC) 68000

MICROTEK PASCAL 8085

OMSI PASCAL AMD 2901

PL/M-80 AMZ 8002

ROLM FORTRAN MACRO-11

SKC FORTRAN RCA 1802

SKC 3121/3132

MSAT HISTORY

T00LS

1979 PFA

1983-4 SPECIFIC TOOLS

1984-5 FCAP, ACAP, CCAP

1985-6 MSAT

LANGUAGES

HOL ASM

FORTRAN - VARIOUS VERSIONS 8600

JOVIAL 8600>

C MACRO-11

RATFOR ROLM ASM

SKC ASM

SYSTEMS

POSITION LOCATING REPORTING SYSTEM (PLRS)

TEAMPACK

TACTICAL COMPUTER SYSTEM (TCS) TRAILBLAZER

INTEGRATED INERTIAL NAVIGATION SYSTEM (IINS) REMOTELY PILOTED VEHICLE (RPV)

SGT YORK FIRE CONTROL

JOINT TACTICAL INFORMATION DISTRIBUTION SYSTEM (JTIDS)

DEVELOPMENT PHILOSOPHY

- . LANGUAGE TABLE DRIVEN
- . COMMERCIAL DBMS
- . USER FRIENDLY
- . DESIGNED FOR EXPANSION/ENHANCEMENT
- . COMPLETELY DOCUMENTED/MAINTAINABLE
- . VALIDATION OF TOOL AND CONFIGURATION MANAGEMENT

STATIC ANALYSIS FUNCTIONS

INITIAL	
•	

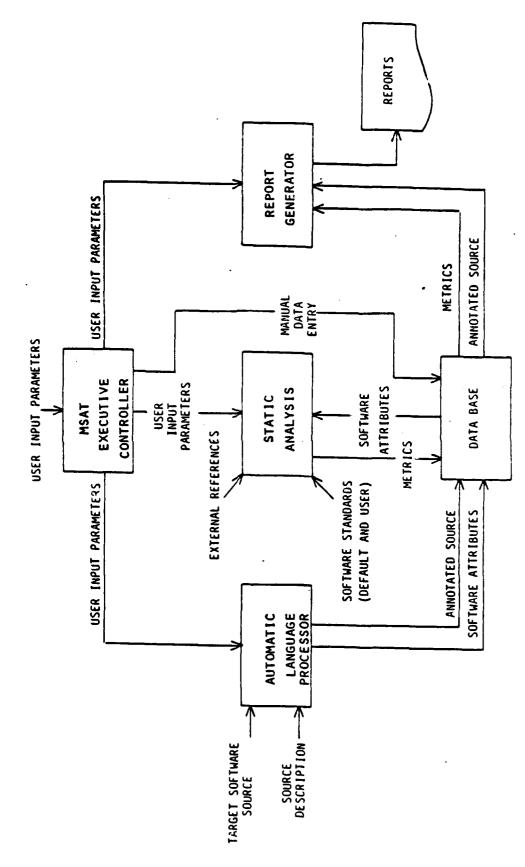
IMPLEMENTATION

- L. AUDITING
- 2. COMPLEXITY MEASUREMENT
- 3. STATISTICAL ANALYSIS
- 4. INTERFACE ANALYSIS
- 5. COMPARISON
- 6. CONSISTENCY CHECKING
- * 7. ERROR CHECKING
- ** 8. STRUCTURE CHECKING
 - 9. COMPLETENESS
 - LO. DATA FLOW ANALYSIS
 - LL. I/O SPECIFICATION
 - L2. CROSS-REFERENCE
 - L3. SCANNING
 - L4. TYPE ANALYSIS
 - L5. UNIT ANALYSIS
- PRODUCED AS A BYPRODUCT OF OTHER FUNCTIONS
- ** REQUIRED TO RETAIN CURRENT TOOL CAPABILITY

VAX/VMS

MSAT SCHEMATIC

S. tr.



MSAT Data/Control Flow

8

CAN.

MSAT OPERATIONAL CAPABILITY

Reports	Standards Compliance	Source Listing/Table of Contents	Ct ange Analysis	Interface Analysis	Software Quality Metrics	Structure Chart	(Error Reports)
SA Functions	Auditing	Complexity Measurement	Comparison	Interface Analysis	Statistical Analysis	Structure Checking	(Error Checking)
Languages	VAX FORTRAN	8085 ASM		(INGRES EQUEL)			

P³I For: Other SA Functions/Metrics, Library of Target System Software Languages

() - Minimal capability for IOC



MSAT REPORTS

- SOURCE LISTING/TABLE OF CONTENTS (TOC)
- . SOFTWARE QUALITY METRIC REPORTS
 - . DETAIL FOR UNIT
 - . UNIT SUMMARY
 - . SYSTEM SUMMARY
- . STRUCTURE CHART
- . ERROR REPORT
- . INTERFACE AMALYSIS REPORT
- STANDARDS COMPLIANCE REPORT
 - . STANDARDS EXCEPTION
 - . UNIT SUMMARY COMPLIANCE
 - . SYSTEM COMPLIANCE
- . CHANGE ANALYSIS REPORT



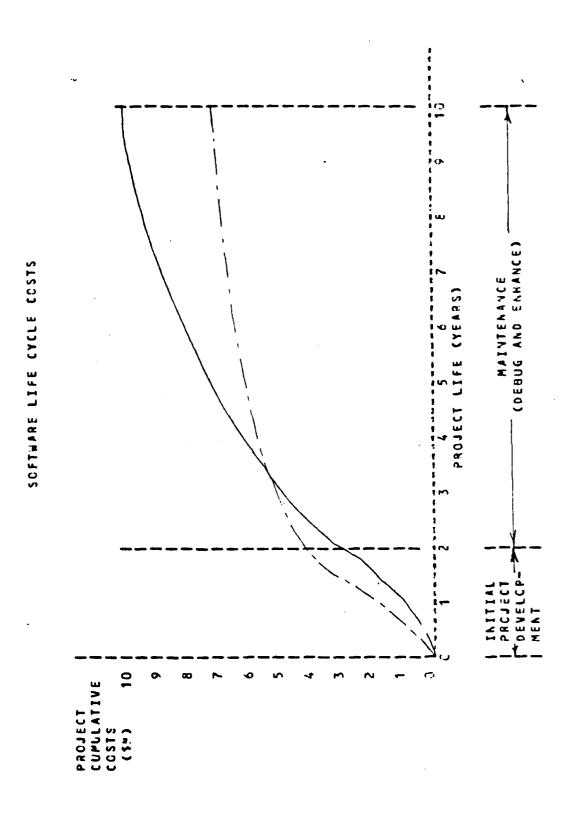
UNCLASSIFIED Output Page : 48

Standards Compliance UNIT SUMMARY Report: MSAT PQT

TSS ID/Version	: MSPQT / 1.0	MSAT Version : 1.0
Cluster Level	: SYSTEM	CSCI Collection Date
CSCI	: n/a	

Standard Document: EPG-STD Language ID : VAX11FOR*	[*] * based on STD document	!
1	# (%) of # (%) of UNITS UNITS UNITS COMPLIANT NON-COMPLIAN	1
[*] Executable Statements <= 200	203 (100.0%) 0 (0.0	
[*] Max Consequetive Executable LOC w/out Comment s <= ']	192 (94.6%) 11 (5.49	(۵
[*] # Entry Points <= 1	203 (100.0%) 0 (0.01	()
[*] # comments (whole + partial) > 0	199 (98.0%) 4 (2.0)	()
[*] Maximum Internal Nesting Level <= 5	198 (97.5%) 5 (2.51	()
[*] % Executable commented >= 80	125 (61.6%) 78 (38.4)	(ا
[*] # Compound Executable LOC = 0	203 (100.0%) 0 (0.0)	i)
[*] # Backward Branches = 0	203 (100.0%) 0 (0.09	()
[*] # Assembly LOC = 0	203 (100.0%) 0 (0.0%	.)
[*] # Prolog lines > 0 (for units with # Exec s	199 (98.0%) 4 (2.0%	,)
tmts > 25) [*] # Prolog lines > 10	196 (96.6%) 7 (3.4%	,)
[*] # Lines of Conditional code = 0	199 (98.0%) 4 (2.0%	.)
[*] McCabe's Cycloma*ic <= 10	182 (89.7%) 21 (10.3%	.)
[*] Myer's Complexity <= 10	187 (92.1%) 16 (7.9%	,)
Total UNITs in CLUSTER =	203	_

UNCLASSIFIED Classification verified by THOMPSON



THE MULTI-LINGUAL STATIC ANALYSIS TOOL (MSAT)

- PROVIDES STATIC ANALYSIS OF THE SOFTWARE
 - AUDITING
 - COMPLEXITY MEASUREMENT
 - STATISTICAL ANALYSIS
 - INTERFACE ANALYSIS
 - COMPARISON
 - ERROR CHECKING
 - STRUCTURAL CHECKING
- PROVIDES QUALITY ANALYSIS OF THE SOURCE CODE
- PROVIDES VISIBILITY INTO THE SOURCE CODE
- PROVIDES CONFIGURATION VERSION COMPARISON

ANTICIPATED MSAT USERS

SOFTWARE DEVELOPERS (SQA)

VERIFICATION AND VALIDATION

DEVELOPMENT TESTERS

USERS (TO UNDERSTAND SOURCE/ALGORITHM)

SOFTWARE SUPPORT CENTERS

SOFTWARE LIBRARIES

FUTURE DEVELOPMENT PROPOSED TO STARS

- A. DEVELOP INITIAL ADA CAPABILITY
- B. MAKE THE SYSTEM TRANSPORTABLE
 - o CONVERT SOURCE TO ADA
 - o ELIMINATE VAX/VMS DEPENDENCIES
 - o STAND-ALONE DBMS
- C. ENHANCE ADA CAPABILITY
- D. DEVELOP A LIBRARY OF LANGUAGE CAPABILITIES
- E. EXPAND THE STATIC ANALYSIS CAPABILITY
- F. EXPAND THE DATABASE FOR MANUALLY COLLECTED DATA

FY86 MSAS GOALS

- o INITIAL CAPABILITY TO ANALYZE ADA SOFTWARE
- OR SIMILAR LANGUAGE*
- o START OF CONVERTING MSAS TO ADA AND MAKE TRANSPORTABLE TO OTHER SYSTEMS

CURRENTLY MSAS HAS CAPABILITY TO ANALYZE FORTRAN AND 8085 ASSEMBLER CODE.

A CLASSIFICATION SCHEME FOR REUSING SOFTWARE COMPONENTS

Ruben Prieto*
Barbara Moore

GTE Laboratories, Inc. 40 Sylvan Road Waltham, MA 02254 (617) 466-2933

Introduction

Software reuse in the context of this paper is the selection, modification and adaptation necessary to fit an existing component into a new software system. The focus of the paper is on the selection problem, i.e., the ability to locate and retrieve an appropriate component from a large collection of components, such as collection of Ada libraries.

A classification scheme is a domain knowledge structure that organizes collections of items to satisfy the needs of the users of the collections. The GTE Reusability project has performed an in-depth investigation on classification schemes with the aim of identifying and adapting one that satisfies the needs of software users. In this paper a faceted classification scheme is proposed. The classes in the scheme are identified by collecting descriptive terms from component descriptions and grouping them by their relationships. The set of collected attributes form a vocabularly of terms that can be used to describe software components by their reusability-related attributes.

The main features of the proposed classification scheme are expandability, adaptability, and consistency. Expandability means that new classes can be added with a minimum of reclassification problems, adaptability means that the scheme can be customized to a particular environment, and consistency means that components from different collections in the same class share the same attributes.

a software-supported query system that facilitates retrievals based on the classification scheme described.

This paper also reports current work on

Library Classification Schemes

Classification is the act of grouping like things together. All members of a group- or class- produced by classification share at least one characteristic which members of other classes do not possess. Classification displays the relationships between things, and between classes of things and the result is a network or structure of relationships which may be used for many purposes.

Classification is a fundamental tool for the organization of knowledge and pervades everyday life from supermarkets warehouses to schools. A library is usually considered as the typical example for classification where a collection has been organized for easy access and retrieval. A collection owes its organization to a classification scheme which in turn is based on a controlled and structured index vocabulary called the classification schedule. The latter consists of the set of names or symbols representing concepts or classes and is listed in a systematic order to display the relationships between classes.

Classification schemes can be arranged in two ways: enumerative and faceted. The enumerative or traditional method is to postulate a universe of knowledge and to divide it into successive narrower classes which will include all the possible compounded classes and arrange them to display their heirarchical relationships. Dewey Decimal classification is a typical example of an enumerative

^{*}Part of this work was conducted at the University of California Irvine in connection with the author's PhD disseratation

decachotomy based hierarchy. All possible classes are predefined.

The faceted method relies not on the breakdown of a universe, but on building up or 'synthesizing' from the subject statements of particular documents. By this method, subject statements are analyzed into their component elemental classes, and it is these classes only which are listed in the schedule; and their generic relationships are the only relationships displayed on its pages. When the classifier using such a scheme has to express a compound class, he does so by assembling its elemental classes. This process is called synthesis and the arranged groups of elemental classes that make the scheme are the facets. Facets are sometimes considered as perspectives, viewpoints or dimensions of a particular domain.

Systematic order in a faceted scheme consists in ranking the facets by citation order according to their relevance to users of the collection. Terms within facets are ordered by their relationship to each other or their conceptual closeness. There are different user selected criteria for ordering terms and by convention, this ordering consists of a one dimensional list where conceptual closeness between any two terms is 'measured' by the number of terms listed between them. When classifying in a faceted scheme, the most significant term in the classification description is a term selected from the facet that is most relevant to the user.

Software Classification

Faceted schemes are very attractive for classifying reusable software because they are, in general, more flexible, precise and better suited classification scheme for reusable software components has been proposed by one of the authors (Prie85). The scheme proposes a component description format based on a standard vocabulary of terms, imposes a citation order for the facets, and provides a conceptual metric to measure conceptual distances between terms in each facet for a more effective selection of closely related items. The scheme is based on the criteria that collections of reusable components are very large and continuously growing, and that they are, even in very specific classes, large groups of very similar components.

An experimental collection of over 200 program descriptors of modules ranging from 50 to 200 source lines of code was used to derive facets and terms of a preliminary software schedule. Two groups of facets were identified: those describing functionality and those describing the environment, three in each group. It was observed that program descriptions consist of two parts: one describing the functionality (i.e., what it does) and the other describing the environment (i.e., where it does it). Implementation details or realization (i.e., how it does it) were not usually included in a description. So, function and environment were selected as facets and realization characteristics used as discriminating factors to separate similar components.

It was observed that functionality equivalent components performed essentially the same function and that differences in their realizations could be identified indirectly through some intrinsic characteristics like size, complexity, and programming language used. Implementation differences based on intrinsic factors are approximate and valid only when the number of functionally equivalent components is large.

Functionality - The three facets for functionality were identified by observing the imperative nature of statements describing functions. e.g.,

<input, character, buffer>,
<substitute, tabs, file>,

< search, root, B-tree>

Description of functionality therefore consist of

<function, objects, medium>

where function is a term naming the function, objects identifies the objects manipulated by the function and medium identifies the 'locus' of the action, usually a data structure.

Environment - The facets for environment were identified as:

<system-type, functional-area, setting>
where system-type is an application independent name typically given to the basic

function described in functionality. For example, report-formatter, scheduler, retriever, expression-evaluator, etc. Functionalarea describes a particular identifiable application dependent function. It is usually defined by an established set of procedures in an area of application like general-ledger, costoperating-system, Setting etc. describes the location where the application is exercised. It captures details of how to conduct certain operations. These environment facets, reflect to some extent, the nature of the experimental sample used but collections in other domains could turn up with other facets such as type of security, accessibility, or design methodology used. A descriptor (i.e., classification code, call name) for a program consists in defining a term from each facet as in:

<substitute/backspaces/file/text_formatter/
program_development/software_shop>

Conceptual Closeness - An important feature of this scheme is the introduction of a conceptual graph to measure closeness among terms in a facet. A conceptual graph is an acyclic directed graph that relates every term in a facet through a set of weighted edges. Terms are at the leves and the nodes are 'super types' that denote general concepts. Weights are user-assigned; that is, the closer the user perceives a relationship of a term to a supertype, the smaller the weight. The example in figure 1 shows a partial weighted conceptual graph for some function names. The notions or supertypes are all related to the notion of function (*) which is the facet name. Closeness is then measured by the closest path between any two terms; for example, measure is closer to add (i.e., 6) than to move (i.e., 16). A reuser perspective was used for weight assignment in this particular graph.

One practical application of a closeness measurement happens during retrieval. If a particular term in a query does not match any available descriptions in the collection, the system then tries the next most closely related term to retrieve descriptions of closely related items.

An abstract view of the scheme is presented in figure 2. Each component (a)

has a descriptor (d) which is an ordered set of terms from each facet (F). Every term in a facet is related to one or more supertypes by means of a weighted conceptual graph. During retrieval, a query is a valid descriptor d of terms selected from each facet. If there is no match in the collection for d then closely related terms are selected by computing distances in the corresponding conceptual graph to make new descriptors d 2 < i < n. Matches on subsequent d's will retrieve components closely related to components described by d.

It is assumed that components require some modifications before being used in a new application based on the fact that code is very specific and an exact match between requirements and available features is almost impossible. Code is the distilled product of several design decisions for which there is usually no documentation unless the whole refinement process from specification through design through code was captured. Even in these ideal circumstances, the refinement process is so long and involved that its mere analysis and understanding would overcome any reuse effort. Understanding of component characteristics through indirect means is therefore essential.

Current Work

The faceted classification scheme, the conceptual distance model and a mechanism to evaluate and rank functionally equivalent components were integrated into a prototype library system. An SADT level 0 diagram of the library system for reusable code fragments is shown in figure 3.

This library system can be seen as a group of procedures that help in query construction and in the evaluation of the retrieved sample for potential reusability. The data base of component descriptors is considered to be the catalog.

The query system (boxes 1.2, and 3) makes use of the classification scheme to interactively generate component descriptors (groups of valid terms used to describe a component). The system guides the user in selecting valid terms from the classification schedules and enforces a citration order for the terms based on the established relevance

order of the six previously defined facets. A query is a six-tuple descriptor of a component. A query may be modified by insertion or removal of terms in a prescribed order (from less to more relevant) resulting in a specialization or generalization of the query.

A query may also be expanded. Queries of closely related terms are constructed based on their conceptual distance. Conceptually closer terms are selected first for the new queries. Groups of queries are ordered by their relevance to the original query. The result is an ordered set of queries from most to least 'related' to the original query. Scope of expansion is controlled by the user. Expansion is used when the original query returns the empty set. Query expansion is central to the library system.

Retrieval (box 4) is implemented by a relational data base system where each program descriptor is a tuple in the database with pointers to source code, documentation and other relevant information. Evaluation (box 5) is a system of its own that normalizes reusability related metrics and ranks the sample according to the estimated reusability effort required to reuse the components.

The evaluation system is based on the assumptions that the collection of components is very large; that several components in a given class of components in the collection may be functionally equivalent; and that there is a need to assist the user in selecting, from among all functionally equivalent components, the one easiest to reuse. The evaluation mechanism estimates potential reusability effort based on four basic

software metrics and on reuser experience. Reuser experience is used as a modifier for the other metrics to adjust their relevance.

Tests with the library system showed better retrieval performance in terms of user satisfaction than regular relational data base systems. Because of the relatively small size of the collection and the limited number of participants, the results, although very encouraging, are only indicative at this time rather than conclusive but work is on the way to scale up the collection and test the system in a production environment.

Effort is under way to implement these concepts of software classification and reusability in the domain of information management software at GTE. Reusable assets has been the focus and a preliminary analysis of the assets domain has resulted in the definition of four basic facets; asset-type (e.g., software, hardware, information). application-area (e.g. business, telecommunications, systems), complexity-level (e.g. code-fragments, subcomponents, modules, subsystems), and reuse-mode (e.g. modify, adapt, use-as-is, call, insert). Work is under way to expand facets and populate the schedules with appropriate terms for each facet. Terms will be defined from the analysis of a representative sample of assets and asset descriptions. A library system for asset management will be the logical follow up in this project.

[PRIE85] Prieto-Diaz. R. A Software Classification Scheme, Ph.D dissertation, Department of Information and Computer Science, University of California, Irvine, 1985

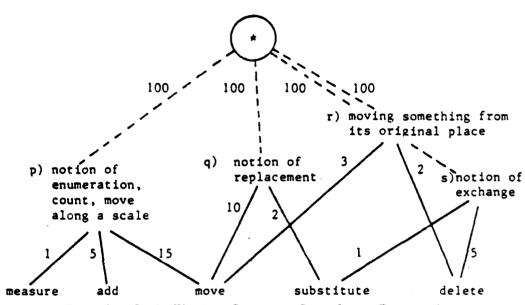
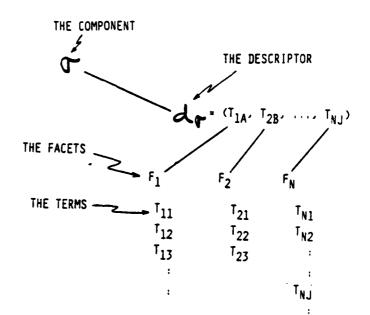


Figure 1. A Partial Weighted Conceptual Graph for the Function Facet



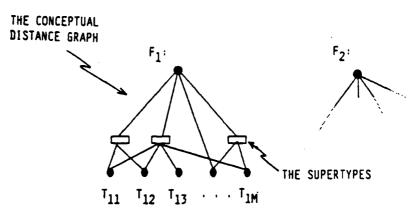


Figure 2. Abstract View of the Proposed Classification Scheme



W.

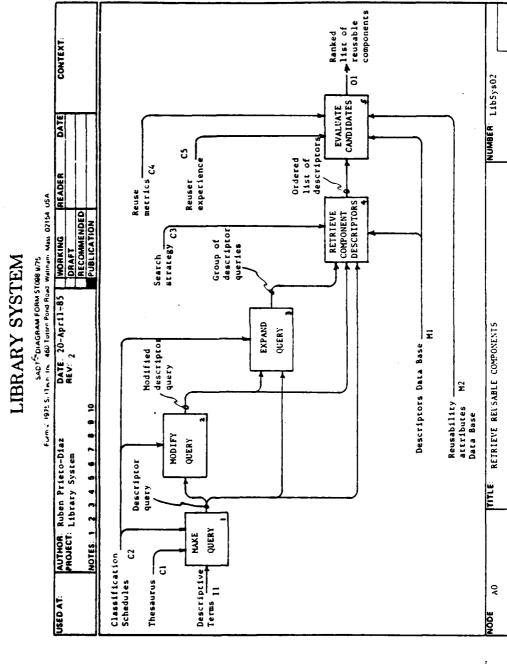


Figure 3. SADT Level 0 Actigram for a Library System

A SOFTWARE CLASSIFICATION SCHEME

Rubén Prieto-Díaz

Sofware Reuse Project Computer Science Laboratories GTE Labs.

October 7, 1985

CLASSIFICATION SCHEME:

- 1- Definitions
- 2- Classification Schemes
- 3- Faceted Schemes
- 4- Our Scheme

DEFINITIONS

Classification: Discovery and display of concepts and

their relationships.

Example:

Eagle -- bird of prey, day hunter Owl -- bird of prey, night hunter

Classification Scheme: Tool for arranging concepts and

relationships in a systematic order based in a controlled

index vocabulary.

Index Vocabulary: Ordered set of names which

represent concepts.

Examples:

vertebrates-amphibians-frogs-toads

000 Generalities

100 Philosophy and related disciplines

200 Religion

300 The social sciences

400 Language

500 Pure sciences

600 Technology (applied sciences)

700 The arts

800 Literature

900 Geography and history

Relationships:

Hierarchical → Indicate subordination or inclusion.

(animal-vertebrate-birds-eagle-golden eagle)

Syntactical → Between terms in classes defined by one or more characteristics.

migratory birds birds of the sea shore the respiration of birds

Classification Schemes:

Faceted → Based on synthesis

Enumerative → Based on exhaustive listings

Facet → A perspective

EXAMPLES

Facets -- by effect on man
by habit
by habitat
by land form
by ground cover
by latitude
by element
by zoologist taxonomy

Domain - animals/fauna

A Faceted Scheme:

(process facet)
Physiology
Respiration
Reproduction

(animals facet)
(by habitat subfacet)
Water animals
Land animals
(by zoologists' taxonomy subfacet)
Invertebrates
Insects
Vertebrates
Reptiles

An Enumerative Scheme:

Physiology Respiration Reproduction

Water animals
Physiology of water animals
Respiration of water animals
Reproduction of water animals
Land animals
Physiology of land animals
Respiration of land animals
Reproduction of land animals

Invertebrates

Physiology of invertebrates
Respiration of invertebrates
Reproduction of invertebrates
Water invertebrates
Physiology of water invertebrates
Respiration of water invertebrates
Reproduction of water invertebrates
Land invertebrates
Physiology of land invertebrates
Respiration of land invertebrates
Reproduction of land invertebrates

Insects

Physiology of insects
Respiration of insects
Reproduction of insects
Water insects

etc....

FACETED vs. ENUMERATIVE SCHEMES

• ENUMERATIVE

- + Extensive
- + Usually incomplete
- + Rigid
- + Central hierarchy

• FACETED

- + Brief
- + High resolution
- + Amenable to automation
- + Flexible

FACETED SCHEMES

• Facet Ordering

Animal Facets		Relevanc	ee	
Pespective	more-		···	less
Zoologist	taxonomy	habitat	habit	effect on man
Marine Biologist	habitat	taxonomy	habit	effect on man
Environmentalist	effect on man	habit	habitat	taxonomy

• Term ordering → Display relationship by lingear ordering

mercury	solitary animals
venus	herd animals
earth	social animals
mars	etc
etc	

• Synthetic Classification

a) Salt water fish -----> fish/marine
b) Frogs of the lake ----> frogs/lake
c) Butterflies of the river ---> butterflies/river

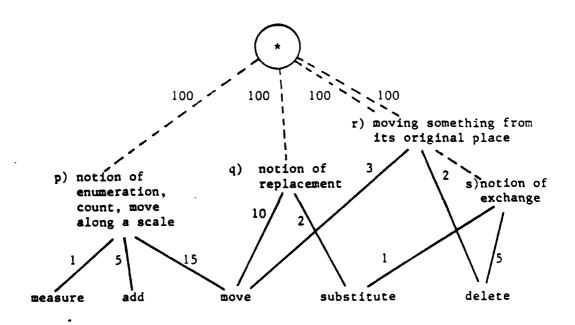
OUR SCHEME → Faceted

- Precision on software component descriptions
- Expandable
- Flexible
- Provides metric for closeness of relationships

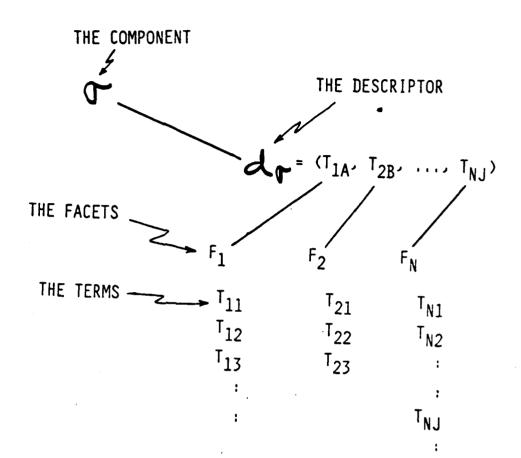
Metrics:

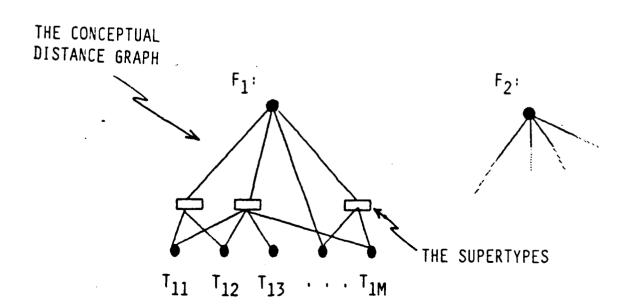
- Facet level → Relevance between facets

 (facets ordered from most to less relevant)
- Term level → Use of user defined supertypes



ABSTRACT VIEW OF THE SCHEME





(VXV

IMPLEMENTATION:

- 1- Observations
- 2- Facet Selection
- 3- Synonym Control
- 4- Component Evaluation

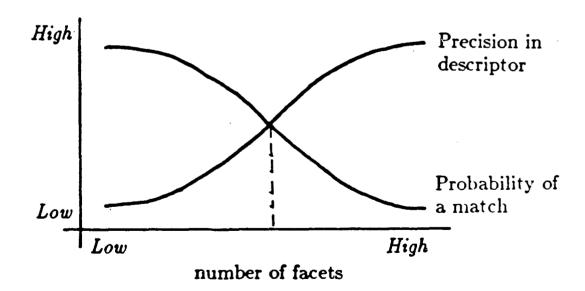
OBSERVATIONS:

£7.

• Component Descriptions → Syntactical relationships among terms

add file to archive read lines from a file convert string to floating number

• Tension Problem on description detail



PARTIAL SCHEDULE:

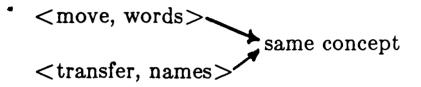
FUNCTIONALITY

(citation order $\rightarrow \rightarrow \rightarrow \rightarrow$)

{function} $\{objects\}$ {medium} input tabs keyboard backspaces output mouse move digits sensor characters append printer insert patterns display tokens extract cards substitute integers tape delete reals disk compare words speech parse strings file decode lines table search buffers buffer files stack measure split tables list lists tree trees

SYNONYM CONTROL

Why? People have different interpretation for different terms.



Need to unify descriptors under same concept

SOME FUNCTION SYNONYMS

GIVEN	NAME	SYNONYMS

input data_entry/scan/enter/read/

output data_output/print/echo/show/write/display/

move transfer/copy/

append affix/attach/concatenate/join/

insert include/push/

extract pick/

substitute replace/exchange/transliterate/

delete remove/erase/cancel/

compare test/

parse recognize/

decode multi_way/muti_branch/selector/

search look_up/find/

measure count/advance/size/ split separate/break_up/

COMPONENT EVALUATION

- Attribute Selection Criteria
 - Validated metrics
 - Objective
 - Easy to use
 - Related to program understanding
- Selected attributes
 - Program size → Source lines of code
 - Program complexity → Conditional statements

THO

- Programming language → Similarities between source and target languages
- Documentation → Quality score

METRIC NORMALIZATION

Why? Ability to rate and rank similar components

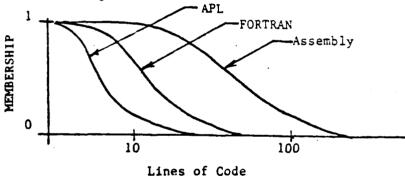
Problem: Attribute metrics function of other factors

Introduction of memebrship functions from fuzzy set theory

EXAMPLE

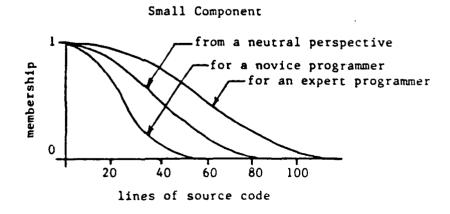
Attribute: Component Size

Measure degree of membership to the class of small components



Role of Reuser experience:

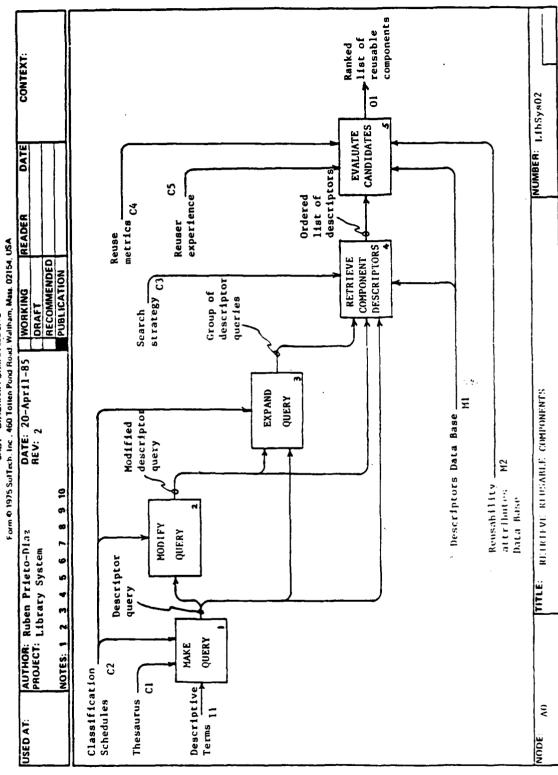
modifier of membership functions



LIBRARY SYSTEM

AND SECRETARIO DESCRIPTION OF SECRETARIO SEC

SADT DIAGRAM FORM STORB 9775 Form © 1975 Sufferb. Inc., 460 Totten Pond Road, Waltham, Mass. 02154, USA



र्गुङ

3(3)

~^

SUMMARY OF CONTRIBUTIONS

- An expandable and adaptable scheme for software classification.
- An approach to measure closeness among terms in faceted schemes.
- A process to define facets and introduction of six reuse related facets.
- Introduction of six reuse related attributes and their metrics.
- Ability to normalize reuse related metrics by using fuzzy functions.
- These concepts can be integrated into a library system as demonstrated by prototype.

GUIDELINES FOR WRITING REUSABLE ADA (R) SOFTWARE

Rick St. Dennis

Honeywell Inc.
Computer Sciences Center
1000 Boone Avenue North
Golden Valley, Minnesota 55427

ABSTRACT

Software reuse is key to significant gains in programmer productivity. However, to achieve its full potential guidelines for writing reusable software must exist and be followed. While language independent, measurable characteristics of reusable software can be the basis for these guidelines, the guidelines themselves should be language-specific. This paper describes ongoing research at the Honeywell Computer Sciences Center to define a set of characteristics of reusable software as well as guidelines for implementing them in the Ada language.

Keywords: Ada, reusable software parts, reusability, object-oriented programming.

Alle to a continued and an all of the U.S. Continued (AIRO)

Ada is a registered trademark of the U.S Government (AJPO)

This work was supported in part by the Office of Naval Research under contract number N00014-85-C-0666.

1. Introduction

(....

Both software production costs and the amount of new software produced annually are skyrocketing. In 1980, the U.S Department of Defense (DoD) spent over \$3 billion on software. By 1990, their expenses are expected to grow to \$30 billion/year [HOROWITZ84]. If current development trends continue, future costs will be increased even more by unreliable software, software delivered late, and continuing maintenance problems.

Today's software needs outpace our ability to produce it, as shown by the backlogs in MIS departments nationwide, and needs are growing each year [STARS83]. There is and will continue to be a serious shortage of qualified programmers to meet these needs. One might expect productivity increases for programmers to make up for at least a part of this shortage. However, software development has been relatively

small year-to-year productivity increases as contrasted with dramatic increases in hardware fabrication [HOROWITZ84]. We feet that a key to significant gains in programmer productivity lies in the area of software is an exponential function of its size. Halving the amount of new software built will more than halve the cost of building the software that we need [JONES84].

Software reuse is an important part of the RAPIER (Rapid Prototyping to Identify End-User Requirements) project for many of the same reasons it is important to software productivity increases in general. One of RAPIER's main goals is "...to develop a prototype engineering environment [that will provide tools and techniques for developing modifiable prototypes q_ickly and inexpensively..." [RAPIER86] The approach to achieving this goal is to build prototypes from reusable software parts. It is the characteristics of these reusable software parts that will provide the modifiability, and the rapid and

inexpensive development of prototypes that RAPIER requires.

To date, no adequate characterization of what makes software reusable exists. It is quite common to read unmeasurable, qualitative admonitions as to what makes software reusable and/or specific examples of software that is claimed to be reusable. However, these admonitions (or "metacharacteristics") and software examples are not enough. Measurable characteristics of reusable software are needed as well as specific guidelines to implement them in source code. Only through use of these characteristics and guidelines can the full potential of reusability be achieved.

The RAPIER project has developed Version 1.0 of "A Guidebook For Writing Reusable Source Code in Ada (R)" [STDENNIS86], [RAPIER86]. This guidebook contains three reusability metacharacteristics, fifteen measurable characteristics that realize the metacharacteristics, and 63 guidelines for implementing these characteristics in Ada source code. Guidebook chapters are organized to follow the Ada Language Reference Manua! [DOD83]. Version 1.0 of the guidebook contains selected chapters covering all major Ada program units, program structures, compilation issues, and visibility rules. Example Ada modules that were written following the guidelines are also provided. This guidebook provides the RAPIER project with a basis to begin writing reusable Ada software parts to be used in its prototyping system.

PRODUCED BY CONTROL BY

This paper outlines the approach to achieving reusability we prescribe in our guidebook. In it we list our reusability characteristics, highlight one characteristic, and provide guidelines supporting it. We also provide example Ada modules written following the guidelines, discuss the relationship between our reusability guidebook and the STARS Application Area, and outline plans for future work.

2. Our Approach To Achieving Reusability

Our approach to reusing source code centers around reusable components, written as Ada packages, classified for both browsing and retrieval, and residing in a library or

software base. See Section 5 of [RAPIER86]. We believe that the features of the Ada language combined with a set of software design and coding guidelines supporting characteristics of reusable software will enable creation and reuse of software in a manner not possible with most other languages and systems. These guidelines will constrain how Ada software is written for the sake of reusability.

Companion work at Honeywell's Computer Sciences Center is also addressing the organization and composition principles that will provide a framework for reuse of components. A classification of components according to behavior has been proposed in Section 5 of [RAPIER86]. Program composition using an adaptation of the operational paradigm for program design has also been proposed in Section 3 of [RAPIER86]. A high-level language for composing programs of components drawn from a software base using a Prototype System Description Language (PSDL) is being designed by International Software Systems, Inc. (ISSI) [ISSI86]. So the characteristics and guidelines in our guidebook fit into an overall approach to reusability.

3. Reusability Metacharacteristics

We propose these metacharacteristics of reusable software:

 Candidate software for reuse must be able to be found.

Findable software must comprise both code and specification. At a minimum, the specification tells users what a software part does, thus allowing them to decide whether it meets their functional needs. A specification may describe attributes of the software part such as author, hardware dependencies, execution time on a particular configuration, and so forth which further assist users in deciding what software is appropriate.

The apparatus for storing and managing software contributes greatly to its findability. That apparatus includes a software base management system and intelligent schemes for classifying software so that searches into the



software base are successful without being frustratingly long.

It must be significantly less costly to find software and reuse it than to recreate it. Both the specifications and the apparatus for managing the reusable software must support relatively low (human and machine) overhead for storing software and searching for it.

Once found, software must be understood enough to be reused.

This requirement involves both the software part's specification and, if its code is to be modified, the way in which it is coded. There are judgments to be made about what attributes of a software part reusers need to know in order to decide whether the software meets their needs.

If the software is to be modified, it must be engineered so that reusers can examine the code and make changes that do introduce errors or unwanted side effects, and that do make the desired alterations.

(3) Once found and understood, it must be feasible to reuse the software.

Software that can be reused-

- o is built for reuse constructed under the constraint that it will be reused.
- o is fit for reuse (i.e., is a "plug-compatible" part)-composable with other code in such a way that it neither interferes with that other code nor allows itself to be interferred with.
- o displays conceptual clarity or appropriateness - presents a useful abstraction (such as a table, a database, a sensor or a stack) at an "appropriate" level.

Each of the software characteristics listed in Section 4 is a means of achieving one (or some) of these metacharacteristics. Figure 1 below relates each of the proposed characteristics to the metacharacteristics it promotes.

4. Reusability Characteristics

4.1 Criteria For Reusability Characteristics

The reusability metacharacteristics in Section 3 are qualitative "good practice"

admonitions. In general, the characteristics listed in this section are measurable or judgable qualities that software should possess in order to meet the metacharacteristics. We have proposed characteristics that are statistically measurable or judgable today or will be measurable/judgable once we have more experience with reusable software. For example, today we can measure if software is free from hidden side effects. However, we cannot judge whether software has the right balance between generality and specificity. Only when software has been reused for some time, we will be able to judge this quality.

The characteristics listed below are also reuse-specific; using them will produce software that is designed and coded a priori for reuse. "Good" software engineering practices will contribute to reuse but will not specifically make software reusable.

Our guidebook only briefly discusses an important aspect of reusability domain or application specificity. We expect that application specificity will be a major factor in enabling software reuse [FRANKOW-SKI85B]. However, just as all software intended for reuse must be built using good software engineering practices, it must be built using application neutral basic reusability guidelines in addition to application specific guidelines. The characteristics listed below are those underlying guidelines for reusability across application areas.

In our guidebook, we post 15 language-independent characteristics of reusable software. For the purposes of this paper, we list all characteristics and highlight #4: Component is designed as object-oriented; that is, packaged as typed data with procedures and functions which act on that data.

4.2 List of Characteristics

- (1) Interface is both syntactically and semantically clear [STANDISH84]
- (2) Interface is written at appropriate (abstract) level.
- Component does not interfere with its environment;.

(4) Component is designed as objectoriented; that is, packaged as typed data with procedures and functions which act on that data.

An object orientation to code involves mapping of "solutions" to our human view of the "problems" the software is trying to solve [BOOOCH83]. Our human view involves objects, attributes of these objects, and operations on objects expressed in a noun/verb sense in English. An object-orientation to software aids understandability since solutions to problems are expressed in our "human terms.

Reusable software should act on objects explicitly. What we are advocating here is a clear definition and method of "acting" on objects. All actions or operations on objects should be defined as subprograms (or their equivalent) with the objects as parameters. Furthermore, the objects or at least their types should be "packaged" as close to the definition of the operations on them as possible. Ideally, they should be packaged together to ease location, reference, and use. To promote reusability it is better not to use global data that is changed implicitly by routines to which it is visible but to pass the data to routines as parameters making it explicit (1) these routines actors/operators on the data and (2) that is just how this data will be treated (e.g., as input only, as a constant, and so forth).

Based on Section 5 of [RAPIER86], we will define operations on data in context as implementations of behaviors that characterize objects, the objects being defined by the set of all behaviors associated with them.

- (5) Actions based on function results are made at the next level up.
- (6) Component incorporates scaffolding for use during "building phase".
- (7) Separate the information needed to use software specification, from the details of its implementation, its body.

- (8) Component exhibits high cohesion/low coupling [BERGLAND81].
- (9) Component and interface are written to be readable by persons other than the author.
- (10) Component is written with the right balance between generality and specificity [MATSUMOTO84].
- (11) Component is accompanied by sufficient documentation to make it findable.
- (12) Component can be used without change or with only minor modifications.
- (13) Insulate a component from host/target dependencies and assumptions about its environment; isolate a component from format and content of information passed through it which it does not use.
- (14) Component is standardized in the areas of invoking, controlling, terminating its function [FONES84], error-handling, communication and structure [LANIERGAN84].
- (15) Components should be written to exploit domain of applicability [NEIGH-BOR84]; components should constitute the right abstraction and modularity for the application.

Figure 1 relates each of the proposed characteristics to the metacharacteristics it promotes.

5. Reusability Guidelines

In this section we provide 7 Adaspecific guidelines from our guidebook that support reusability characteristic number 4 pertaining to object-oriented software. [FRANKOWSKI86A] also discusses use of an object-oriented paradigm to build reusable Ada software.

5.1 Context For Guidelines

There are, in general, two kinds of reusable software parts - directly reusable parts and indirectly reusable parts. Directly reusable parts are those whose behavior or effect is catalogued, that is, "advertised" in

the catalog(1) of reusable software that developers use to determine what software parts are available for reuse. Directly reusable parts are what developers search for and choose. Indirectly reusable parts support directly reusable parts; they provide the environment, the ancillary definitions and data that the directly reusable parts need in order to perform correctly. In the ideal case, indirectly reusable parts are incorporated into the program under construction automatically by a software base management system.

(1) A catalog can be an automated software repository's classification scheme, a list of component names and descriptions on paper, or even a rumor the developer hears from a colleague down the hall.

Reusable parts should be objects. As abstractions, objects have properties (data) and allowable operations on this data. The Ada package should be the realization or concrete implementation of the object abstraction. Types and data objects/ variables implement data; subprograms/tasks implement operations. Packages bundle these things up nicely.

5.2 Sample Guidelines

The following guidelines taken from our guidebook prescribe how to write reusable Ada software satisfying an object-oriented paradigm. Guidelines G10-1, G10-2, and G10-3 provide a specific scheme for writing reusable Ada software in terms of Ada compilation units. Guidelines G6-2, G6-3, G7-2, and G9-2 support this scheme for Ada subprograms, packages, and tasks. We encourage use of generic subprograms and packages in compliance with these guidelines. Please refer to our guidebook for further details on the use of generics.

G10-1: Use library unit package specifications as the encapsulation mechanism for directly reusable software (i.e, data and operations on the data).

Library unit packages are our "unit of reusability" with packages specifications as the standard unit for directly reusable software parts. It is the specifications of operations on data as well as data contained in these packages that are directly reusable. These operations are in effect interfaces to reusable objects. See Figure 2.

G10-2: Only "first level" nested nonpackage entities in library unit package specifications form the basis for "catalogued" directly reusable objects/software.

Ada packages can be nested to any level allowed by a compiler implementation, and nesting can be used as desired for implementing reusable components. However, for each of "cataloging" there should be a practical limit to the level of nesting of packages that encapsulate reusable software. G10-2 simply states that only first-level data and specifications for operations on data form the basis for reusable software and are "catalogued". Data and operations within nested packages are not catalogued as reusable even though they are accessible to client programs according to the Ada language definition. Nesting can easily complicate the environment or context for reusable software. For example, nesting provides an environment for declaration order information hiding, and visibility rules which is hard to reuse and to understand, and in which operations and data are hard to classify. Classifying only entities that are visable at the first level as reusable operations on data in context will avoid this complication.

G10-3: Use secondary unit package bodies, package specifications containing only data, and subunits corresponding to first-level package body nested stubs as the encapsulation mechanism for indirectly reusable software.

This guideline, along with G10-1, states that all reusable Ada software should be written in terms of packages. In particular, subprograms (with the exception of main subprograms) and tasks should be written either directly within the declarative parts of library unit packages or in that context through the use of body stubs. In Ada, main programs must not be contained in packages. However, we do not treat them as reusable. It is the library unit packages they reference that are reusable. Secondary unit (library unit) package bodies are indirectly reusable.

Subprograms and tasks in the context of secondary unit packages (e.g., package bodies) are indirectly reusable. Library unit package specifications containing only data are indirectly reusable as well. See Figure 2 to clarify the distinction between directly and indirectly reusable software parts.

G6-2: All reusable subprograms except a main program must be written within a library unit package.

In view of guidelines G10-1 and G10-3 reusable subprograms must be written in packages. These packages and their contents are the reusable software in a software repository; they are "glued" together by a main program which is invoked from the environment. If this gluing is automatic or easily specifiable in a very high-level-language, main programs do not have to be kept in a repository. It is the reusable parts that they glue together that are important. However, if a main program glues together a "system" which can be viewed as a potential component of other systems, then that program should be put in a package which will be catalogued as directly reusable software and should be called from "another main program".

G6-3: Use subprogram declarations to specify interfaces to reusable objects. Use subprogram bodies to implement these interfaces and properties of the objects.

The interfaces to reusable objects specified in subprogram declarations comprise a name, parameters of particular types and modes, and return types for functions. Subprogram bodies contain the executable code fo. rausable objects. This code performs useful work. We are saying that the use of both subprogram declarations and bodies is important. The only exception to this guideline is a main program callable from the environment rather than by other software. In this case, a body alone is sufficient. This guideline is related to G7-2 prescribing that package specifications implement interfaces to object abstractions and their bodies implement specific details of these abstractions.

G7-2: Use package specifications to specify the interface to object abstractions; use package bodies to encapsulate implementationspecific details of these abstractions not needed by client software.

Simply stated, decide what object abstraction a package should implement, decide what the interface to this abstraction should be, and implement these as visible specifications for operations on data in the public part of a package specification. Decide what the implementation structure of the abstraction should be and implement this and all other details in the private part of the package specification and a corresponding package body. This separation benefits the package itself and its environment. The less "connection" a package has with the outside world (e.g., the smaller the visible part of a package specification), the lower its coupling with other modules. Once modules in a package's environment begin to depend on particular visible entities that really should have been hidden, the package becomes less and less insulated from its environment.

There are two strategies for providing abstractions as reusable objects [BOOCH85]. These are: (1) using packages to implement abstract data types and (2) using packages to implement abstract state machines.

- (1) Abstract Data Types: Provide the basis for multiple "public" reusable objects with common operations on implementations of the operations in corresponding package bodies. The object abstraction can then be reused by client software (multiple times) by declaring variables (external) to the package and using the operations provided by the package to manipulate these variables.
- Abstract State Machines: Provide single sharable, "private" reusable objects and operations on these objects. Do this by encapsulating types of reusable objects in package bodies. This limits client software from declaring and using multiple instances of the reusable objects since their types are hidden. Provide specifications for operations on reusable objects in package specifications. Provide variable declarations for the reusable objects and implementations of operations on the objects

in the case where the types of the reusable objects are not "composite". These operations may contain parameters if the types of the reusable objects are composite," and "atomic" public types from which these types are constructed are declared in package specifications. Client software can only reuse the specific instances of object abstractions contained in these packages. This software can only indirectly access the variables implementing reusable objects through interfaces provided by visible subprograms specified in the package specifications.

G8-2: Use task declarations to specify interfaces to reusable objects. Use task bodies to implement these interfaces and properties of the objects.

This guideline is similar to guideline G6-3. For tasks, as compared to subprograms, interfaces are concerned not only with parameter passing but also with synchronization. While subprograms can optionally have a separate declaration and body, tasks must have both declarations and bodies. Tasks and their entries, just as subprograms, should be treated as interfaces to reusable objects.

6. Example Ada Modules

The example Ada modules below are taken from the design of a reusable software repository developed at the Honeywell Computer Sciences Center. This repository supports retrieval, submission, and maintenance of categories of inventory items stored in a database management system. Its user interface is menu oriented. Specifically, the modules below are:

a package specification for the repository Menu_Manager,

- (2) a package body for the repository Menu_Manager, and
- (3) a procedure subunit for one of package Menu Manager's nested subprograms, Create Initial Menu. The object abstraction implemented in this package is a menu and associated menu stack with operations including Create_Initial Menu, Display Menu, and Process Menu-Response. Package Menu Manager implements an "abstract data type" by exporting menu-oriented types and operations. It also implements an "abstract state machine" in that it contains a nonexportable stack of menus in its body.

In the examples, package specification Menu-Manager and the type and procedure declarations contained in its visible part are directly reusable. Its private part type declarations are indirectly reusable. Package body Menu-Manager is indirectly reusable as is its nested data declarations and subprograms. The subunit for procedure Create_Initial_Menu is indirectly reusable even though it is compiled separately.

These example modules are provided primarily to illustrate use of our "object-oriented" guidelines to write reusable Ada software. The modules also illustrate other guidelines contained in our guidebook, most noticably, those pertaining to a standard form for reusable software parts.

EXAMPLE: MODULE 1

with DATABASE_INTERFACE; package MENU_MANAGER is

- -- Revision History: Created 2/20/86 P. Stachour
- -- Purpose
- -- Explanation: Provide data structures for and operations on
- repository menu objects.
- -- Keywords: menu, menu_manager
- -- Associated Documentation: Design for Honeywell Reusable Software Repository
- -- Diagnostics:

MENU_MANAGEMENT_ERROR : exception;

- -- Packages: None
- -- Data Declarations:
- -- Types:

type MENU is private;

type MENU_NUMBER is range 1..100;

type MENU_ITEM is range 1..55;

- -- Objects: None
- -- Operations: Subprograms:

procedure CREATE_INITIAL_MENU (M_NUMBER : out MENU_NUMBER);

- -- Purpose:
- -- Explanation: Create initial repository menu.
- -- Keywords: initial_menu, create_initial_menu.
- -- Parameter Description:
- -- M_NUMBER: Menu number associated with initial menu.
- -- Associated Documentation: same as above.

procedure DISPLAY_MENU (M_NUMBER : in MENU_NUMBER);

- -- Purpose:
- -- Explanation: Displays specific menu.
- -- Keywords: Display_menu.
- -- Parameter Description:
- -- M_NUMBER : Number of menu.
- -- Associated Documentation : same as above

procedure PROCESS_MENU_RESPONSE (M_NUMBER : in MENU_NUMBER; MENU_ITEM_SELECTED : in MENU_ITEM; EXIT : out BOOLEAN

- -- Purpose:
- -- Explanation: Process response specified by menu selection.

 This processing may involve a call to

 Display_Menu and ACCEPT_MENU_RESPONSE and a
 recursive call to PROCESS_MENU_RESPONSE.
- -- Keywords: menu_response, process_menu_response.
- -- Parameter Description:
- -- M_NUMBER : Number of menu.
- -- MENU_ITEM_SELECTED : Specific item from menu selected.
- -- EXIT : Indication to exit menu system.
- -- Associated Documentation: Same as above.
- -- Tasks: None -- Private:

private

type MENU is ...;

end MENU_MANAGER;

```
EXAMPLE: MODULE 2
with INVENTORY_ITEM, CATEGORY, USER, TEXT_IO;
with USER_STATE, BULLETIN_BOARD, COMMAND_PROCESSOR, FILE SYSTEM,
    SYSTEM SUPPLIED UTILITIES;
package body MENU MANAGER is
-- Revision History: Created 02/21/86 P. Stachour
   Explanation: Provide data structures for and operations on
             repository menu objects
   Keywords: menu, menu_manager
-- Associated Documentation: Design for Honeywell Reusable
                        Software Repository
-- Assumptions/Resources Required: None
-- Side Effects: None
-- Diagnostics: None
-- Packages: None
.sp 0.4v
-- Data Declarations:
-- Types:
   type MENU_ACCESS is access MENU;
   type MENU_STACK_ELEMENT is
     record
      MENU POINTER
                              : MENU ACCESS:
      MENU_FILESYS_LOCATION: STRING (1..100);
      end record;
   Objects:
   MENU_STACK
                      : array (1..31) of MENU_STACK_ELEMENT;
   MENU_STACK_INDEX : NATURAL :=0;
-- Operations:
   Subprograms:
   procedure CREATE_INITIAL_MENU (M_NUMBER : out MENU_NUMBER)
                                   is separate;
   procedure DISPLAY_MENU (M_NUMBER: in MENU_NUMBER) is separate;
   procedure PROCESS_MENU_RESPONSE (M_NUMBER : in MENU_NUMBER;
                       MENU_ITEM SELECTED : IN MENU_ITEM;
                                       : out BOOLEAN)
                                         is separate;
   . -- Other operations on MENU-oriented parameters.
  Tasks: None
-- Initialization:
begin
```



```
exception
when INVENTORY_ITEM INVENTORY_ITEM_ERROR =1
...
when others =1
...
raise MENU_MANAGEMENT_ERROR;
end MENU_MANAGER;
```

EXAMPLE: MODULE 3 separate (MENU MANAGER) procedure CREATE_INITIAL_MENU (M_NUMBER : out MENU_NUMBER) is -- Revision History: Created 2/21/86 P. Stachour -- Purpose: Explanation: Creates initial repository menu by reading data for it from a host file and placing it on the MENU MANAGER menu stack. Keywords: INITIAL_MENU, CREATE_INITIAL_MENU -- Associated Documentation: Design for Honeywell Reusable Software Repository -- Parameter Description: -- M NUMBER: Number of menu created. -- Assumptions/Resources Required: None -- Side Effects: None -- Diagnostics: None -- Packages: None -- Data Declarations: Types: None Objects: FILE_DESIGNATOR:FILE_SYSTEM.FILE_NAME:="DRAO(SOURCE)FILE_NAME.TXT"; -- Operations: -- Subprograms: None Tasks: None -- Algorithms: begin -- CREATE_INITIAL_MENU -- read from host file, create menu, and place on MENU_S1ACK; -- increment MENU_STACK_INDEX by 1; M_NUMBER := MENU_STACK_INDEX + 1; exception when others =1

end CREATE_INITIAL_MENU;

7. Relationship To STARS Application Area

The STARS Application Area, in its series of Application Systems and Reusability Workshops, is working toward definition of a reusability guidebook. This work is being done by four groups: Part Taxonomy/Requirements/Metrics, Incentives, Library, and System/Design Integration. Our work at Honeywell on a reusability guidebook for RAPIER is relevant to the STARS reusability effort in the following ways:

- o Our guidebook can serve as the framework for major sections of the Application Area guidebook;
- o Our reusability characteristics compliment and some areas extend the Part/Taxonomy/Requirements/Metrics Group's reusability model work and specifically define reusable Ada (source code) parts;
- o Our reusability guidelines implicitly provide criteria for choosing reusable Ada software for reuse. They support measurable reusability characteristics and as such can and should be the basis for reusability metrics. This is also relevant to the Part/Taxonomy/Requirements/Metrics Group;
- o Our reusability guidelines provide a methodology for building reusable Ada software which is appropriate to the System/Design/Integration Group;
- o Our reusability guidelines imply a particular cataloging scheme for libraries of reusable software parts and acceptance criteria for these parts before insertion in the libraries. This is relevant to the library Group.

8. Future Work

In the upcoming year we plan to complete the remaining chapters of our reusability guidebook and refine/add to the Ada examples it provides. Guidelines from the guidebook will be used to construct reusable software components for RAPIER's software base management system. We also plan to circulate the guidebook for review. Feedback we receive from RAPIER prototyping

experiments and the guidebook review will enable us to evaluate our reusability characteristics and guidelines and refine them accordingly.

9. Acknowledgements

I would like to thank a number of people from the Honeywell Computer Sciences Center who assisted me in preparing this paper: Jacklyn Lipscomb for her technical editing, Paul Stachour for the example Ada modules appearing in Section 6, and Elaine Frankowski for her reviews.

Bibliography

[BERGLAND81] G.D. Bergland. "A Guided Tour of Program Design Methodologies," IEEE Computer, Vol. 14 No. 10, October 1981, pp. 13-37.

[BOOCH83] Grady Booch. "Object-Oriented Design," Tutorial on Software Design Techniques. Ed. P. Freeman and A. Wasserman, 4th edition (Catalog Number EHO205-5), IEEE Computer Society Press, 1983.

[BOOCH85] Grady Booch. "ACM SIGAda Tutorial: Ada Methodologies," ACM SIGAda Meeting, July 30, 1985.

[DOD83] United States Department of Defense. Reference Manual for the Ada Language: ANSI/MIL-STD-1815A, United States Department of Defense, January 1983.

[FRANKOWSKI85b] Elaine N. Frankowski, Christine M. Anderson. "Design/Integration Panel Report, "Proceedings of the STARS Reusability Workshop, April 1985.

[FRANKOWSKI86a] Elaine N. Frankowski. "Why Programs Built From Reusable Software Should Be Single Paradigm," Proceedings, STARS Applications System and Reusability Workshop, to appear (March 1986).

[HOROWITZ84] Ellis Horowitz, John B. Munson. "An Expansive View of Reusable Software," IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September 1984, pp. 477-487.

[ISS186] International Software Systems, Inc.. "PSDL: Prototype System Description Language," ISSI Technical Report, unnumbered, January 30, 1986.

[JONES84] T. Capers Jones. "Reusability in Programming: A Survey of the State of the Art," IEEE Transactions on Software Engineering, Vol. SE-10 No. 5, September 1984, pp. 488-494.

[LANERGAN84] Robert G. Lanergan, Charles A. Grasso. "Software Engineering with Reusable Designs and Code," IEEE Transactions on Software Engineering, Vol. SE=10D, No. 5, September 1984.

[MATSUMOTO84] Yoshihiro Matsumoto. "Some Experiences in Promoting Reusable Software: Presentation in Higher Abstract Levels," IEEE Transactions on Software Engineering, Vol. SE-10 No. 5, September 1984, pp. 502-513.

[NEIGHBORS84] James M. Neighbors. "The Draco Approach to Constructing Software

from Reusable Components," IEEE Transactions on Software Engineering, Vol. SE-10 No. 5, September 1984, pp. 564-574.

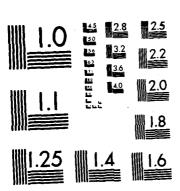
[RAPIER86] RAPIER Project. "Final Scientific Report: RAPIER Project (Contract No. N00014-85-C-0666," Honeywell Computer Sciences Center, Golden Valley, MN, March 1986.

[STANDISH84] Thomas A. Standish. "An Essay on Software Reuse," IEEE Transactions on Software Engineering, Vol. SE-10 No. 5, September 1984, pp. 494-497.

[STARS83] STARS. Software Technology for Adaptable, Reliable Systems (stars) Program Strategy, U.S. Department of Defense, April 1983.

[STDENNIS86] Rick St. Dennis. "A Guidebook For Writing Reusable Source Code in Ada (R), "Honeywell Computer Sciences Center Technical Report, Version 1.0, March 1986.

SOFTHARE TECHNOLOGY FOR ADAPTABLE RELIABLE SYSTEMS
(STARS) MORKSHOP MARCH 24-27 1986(U) NAVAL RESEARCH LAB
WASHINGTON DC MAR 86 UNCLASSIFIED F/G 12/5 NL



MICROCOPY RESOLUTION TEST CHART NATIONAL BUREAU OF STANDARDS-1963-A

ALTERNATIVE TECHNOLOGIES FOR SOFTWARE REUSABILITY

Mark Simos

System Development Corporation
Software Technology
Research & Development Division

Introduction

Beginning with McIlroy's call for a software components industry (McIlroy), discussions of software reusability have been strongly influenced by the compelling paradigm of hardware "parts". While the comparable notion of discrete software "parts" (in the form of programs or subroutines) has certainly had beneficial impact on the discipline of software engineering, it has also tended to limit our conception of reusability in the context of software. An important message emerging from current research and in particular from the STARS Applications Area Workshops is that design for reuse is an essential component of a long-term strategy for software reusability. Yet the processes of modifying, transforming or generating software components are resources just as reusable as the concrete software parts themselves, if these "active elements" of software development can be captured in the form of tools such as program generation systems (Horowitz). This reuse through regeneration effectively fuses the design and "manufacture" of software in ways that have no clear analogies in the hardware sphere. (Note, though, that such developments as silicon compilers and VLSI design libraries are beginning to provide similar flexibility in hardware technology. This suggests that solutions to problems now particular to software reuse should eventually be applicable across the full hardware-software spectrum.)

There is currently much ongoing research on a broad range of technologies, from very high-level languages (VHLLs) to automated software parts composition systems, that could contribute significantly to the overall goal of decreasing the amount of hand-written code needed to implement

application programs (IEEE). SDC's Software Technology Research and Development department is addressing the problems of software reusability with a number of different programs, with particular focus on program generation technology and very high-level application-specific languages, or ASLs. In this report, we will describe some of our experiences with this technology and its implications for reusability. We also touch on techniques from the field of artificial intelligence, an area that has been less discussed with regard to software reusability. The AI (Logic-Based Systems) department at SDC has developed some innovative approaches in the area of knowledge representation and expert systems development that integrally connected with issues of software reuse.

General Approach

Our long-term objective is to develop a methodology for identifying and characterizing potential high-payoff domains for software reuse, and a set of criteria for selecting the appropriate technology, or mix of technologies, to capture the commonality within these domains. We believe that certain technologies for software reuse are best-suited to application domains with particular characteristics. As a starting point, we are working to define an integrating framework, or taxonomony, that usefully discriminates among diverse technical approaches to software reuse, including:

- (1) conventional techniques of reuse (such as ad-hoc reuse and code copying, language features supportive of reusability, and software libraries)
- application-specific languages and application generators

- (3) knowledge-based or expert systems
- (4) various hybrids of the above approaches

Today lists and arrays are standard alternative data structures that one selects for use based on criteria such as time/space tradeoffs and mode of access. As alternative reusability technologies mature, the choice of a library of discrete software parts, an application generator or an expert system will similarly be made on the basis of characteristics and requirements of the intended application domain. After briefly describing our work in these areas, we will offer some tentative conclusions about useful criteria for evaluating the proper 'fit' between application domains and key technologies.

Conventional Approaches to Software Reusability

Conventional approaches to software reusability seek to reduce the amount of code that must be written by hand by isolating fragments of application software as pieces that can be shared and reused by many specific applications. This kind of reuse can occur in zeveral forms, including ad-hoc reuse of code (sometimes called "scavenging"), use of language features that are supportive of software reuse, and software libraries.

There are severe difficulties with ad-hoc reuse (i.e., reuse of a component that was not written with reuse in mind) that nullify most benefits associated with true reusability. It is hard to estimate the amount of time and effort involved in modifying the code for a new application, or, in fact, whether it would be cheaper to write the code from scratch. Subtle discrepancies between the requirements of the original and the retargeted application can lead to decreased reliability or efficiency of the final system. Nevertheless, for some situations ad-hoc reuse may be cost-effective, particularly when there is not much potential for recurring applications, or when requirements of the new application can be modified to accomodate less flexible features of the original application.

Libraries of standard subroutines have achieved some success in certain application domains, such as mathematical routines, graphics packages, or operating system services. There are a number of common features to the applications where this approach has succeeded:

- (1) The libraries are organized around specific application domains.
- (2) There are standard interfaces, calling and naming conventions that effectively make the library a uniform set of services.
- (3) The routines in the library encode operations for which there is a known and fairly standard algorithm. The functionality of the routine does not vary depending on dynamic characteristics of the point of call.

There are a number of critical issues to be faced in developing large libraries of reusable software components, problems such as configuration control quality assurance. cataloguing and documentation. The definition of syntactic and semantic interfaces is one of the main technical barriers to the effective reuse of software (Batz83). Though it is easy to think of software libraries as "current" or certainly "near-term" technology, developing libraries on a large enough scale to really impact productivity will push the state of the art, especially of database technology, as hard as program generation techniques.

Besides the problems stemming from inadequate technology, there are also domains where maintaining a library of concrete software subroutines is not a good fit with the requirements of the domain. One example is simple, low-level functions that require tailoring according to a large number of parameters. If the options for selecting the right version of the routine vary orthogonally, one could quickly wind up with an exponential number of components to be stored in the library. Some form of program generation from specifications is needed for these applications.

Admittedly, many problems associated with ad-hoc reuse or standard subroutine libraries have been addressed by advanced features of Ada*, which was designed with reusability as a clear priority. SDC is directly supporting the Ada movement through the formulation of Ada-based methodologies,

studies in reusable Ada components, development of Ada tools, automatic generation of Ada software from a high-level specification and active participation in the Ada community.

The use of Ada features such as packages, generics, strong typing, default parameters, and tasking to support reusability have been described extensively elsewhere. These features of Ada have significantly extended the domains in which reuse of static software components will be viable. It is interesting to note, for example, that features such as generic program units have shifted functionality onto the Ada compiler that previously would have required program generation techniques. However, because Ada has been standardized, any further extensions to these facilities for generating Ada code at compile time cannot be part of the Ada compiler per se.

Application-Specific Very High-Level Languages

Application-specific languages (verv high level languages that are designed for a particular application area) offer a useful programmatic interface to reusable software modules. Such a language can act as a tailored query language for accessing a repository of reusable algorithms within a narrow domain, as an automated parts-composition system, linking together static routines from a library, or as a parts-generation system, creating instances of a given subroutine optimized to the requirements of each instance of use. Thus it can provide both the flexibility and generality of a highly parameterized set of routines, and the efficiency of tailored code.

ASLs will have highest pay-off in narrowly focused applications areas where many slightly customized versions of a single basic program are created from large, well-understood libraries of basic functions (Standish). ASLs permit the direct embedding of application-specific methodology in the generation system. ASLs can be easier for both programmers and computer-naive application specialists to use than general-purpose high-level languages, because they allow tasks to be specified in a non-procedural language close to the terminology of the application domain. In addition, they

allow typical sequences of actions to be specified at a higher level. Arguments that must be provided explicitly in a call to a library subroutine may be taken implicitly from context in an ASL specification. Also, an ASL processor can perform more extensive static checks for semantic validation than is possible with embedded subroutine calls. Thus, an AS! is a particularly useful interface to a set of services providing access to a persistent data structure such as a database. where there are strict integrity constraints on allowable sequences of operations. The syntax of the ASL can disallow invalid sequences of operations that would have to be defected at run-time if called as a sequence of subroutine calls from a general-purpose programming language.

We believe ASLs are a more feasible near-term alternative than very high-level general-purpose specification languages (Cheatham). Because ASLs use application terminology, they are less abstract, hence easier to use and maintain than formal or algebraic specifications.

An ASL is useful by virtue of its close connection with domain terminology of the target domain, its narrow focus, its non-procedural level of specification, and the guaranteed correctness of its transformation. Languages of this sort can serve as the input specification to two kinds of generation systems:

- (1) a highly-parameterized generic application program, which simulates the behavior of many special-purpose applications by performing sophisticated run-time decision-making;
- (2) an application generator, which transforms the specification into a tailored application program in a lower-level programming language.

*Ada is a registered trademark of the U.S. Government. (AJPO)

In practice, the two options are similar, except that the former embeds the generation expertise at compile time, the latter chooses the proper actions at run-time. A compiled implementation, or application generator, might be more suitable for stable applications that will be run with exactly the same

parameters a number of times. Also, since the output of an application generator need not be a program in the ordinary sense, application generators can be useful in generating multiple output files that must be kept synchronized from a single high-level specification. An interpreted implementation is more appropriate for interactive development of specifications and/ad-hoc or onetime usages of the generation system. We refer to both highly parameterized applications and application generators application-specific (ASLs), languages because the use of high-level terminology from the application domain is a common strategy of both approaches.

A Meta-Generator for ASL Systems

many DoD applications. development of application-specific languages is technologically feasible, but the development cost has seemed prohibitive. These development costs are steadily decreasing, however, as compiler specification and generation techniques approach the stage where entire tools in the language-processing domain can be automatically generated from their specifications. SDC has developed a generation system for tool-building known as the Syntax and Semantic Analysis and Generation System (SSAGS), a Ada-based generation system based on attribute grammars (Knuth). Integrated with a standard lexical analyser generator and parser generator, SSAGS accepts and validates an attribute grammar specification of the semantics of a language, and automatically generates a semantic evaluator for the specified language.

SSAGS provides many advantages for the implementation of ASLs. The use of attribute grammars within SSAGS permits the specification of language semantics in a very clear - and hence less error-prone fashion. In addition, SSAGS is based on ordered attribute grammars (Kastens), a restricted class of attribute grammars that allow a language specifiction to be statistically checked for valid semantics. To take full advantage of this static validation, SSAGS functions as an application generator in the sense described above, unlike some interactive attribute-grammar based systems such as the Cornell Program Synthesizer (Teitelbaum81). Thus translators implemented in SSAGS require less interactive debugging and can be maintained from single, reusable specifications. SSAGS has successfully been used to produce several translators, including an Ada-to-Diana translation system and a configuration ASL for Burroughs XE550 systems. We are currently using SSAGS to implement an ASL for message format validation in the message processing domain. In addition, the SSAGS translator itself is specified in and generated by SSAGS. We believe that use of a translator generator system like SSAGS, together with the strategy of defining small languages for narrowly defined domains is a key to the cost-effective implementation of ASLs for reusability.

Expert Systems

In the context of software reusability. domain analysis usually involves examining a collection of application programs addressing a similar class of problems (e.g., air traffic control or business systems) in order to identify potential reusable software components or algorithms (CAMP). It may seem out of place to discuss expert systems technology in this context. Typically, expert systems automate what was previously a largely human activity; domain knowledge is gleaned from human experts and often can't be reduced to deterministic algorithms. Hence, there is less likely to be a body of conventional programs supporting an application domain being considered for expert system support.

But the presence of conventional applications is not a dependable indicator of whether an expert system approach is most appropriate for a domain. A currently unautomated application area may be quite amenable to conventional software techniques (or may not be worth automating at all). Conversely, for some problem domains currently supported by conventional software a significant increase in software reuse may not be feasible through a parts-library or program generation approach alone. Knowledge-based techniques and heuristics may be the level at which commonality can best be factored into the domain.

For example, if choice of the appropriate algorithm for a given problem situation requires extensive semantic knowledge of the

application domain, knowledge representation techniques may be the most suitable way of encoding this knowledge. In another case, the performance needs of the domain may be stringent enough that subroutines, to be usable, must be optimized for the point of use. In such domains, knowledge-based program synthesis or program transformations may be a prerequisite to effective use of software parts. Note that these situations might utilize knowledge-based technology in two very different ways.

- (1) Artificial intelligence techniques and languages may be directly used within the system being developed.
- (2) Artificial intelligence techniques may be used in combination with library access, program generation, parts composition to facilitate reuse of conventional software.

In either case, expert systems provide reusability in ways that are not available with other techniques. (We will avoid discussion of functional or logic programming language features that support reuse, since these advantages would be confined to direct AI applications.)

Knowledge-based technique provide a means of isolating domain commonality at a more abstract level than that of concrete subroutines, or even standard algorithms and procedures. This allows, at least potentially, a separation of procedural from declarative knowledge which is difficult to achieve in conventional programming languages. It has also been difficult to achieve this separation in many "traditional" expert systems, which are implemented as large, unstructured sets of rules combining conditions and actions in a single framework. KNET (Freeman83), a semantic network knowledge representation system developed by SDC, has several features that help to partition semantic and domain-specific knowledge-- the model of the domain-- from the logic of applications making procedural use of that knowledge. SDC has used KNET to implement a large expert system for the automatic configuration of Burroughs computer equipment (Freeman85). The system is intended to support both the product experts (the plant engineers), who create and modify the

knowledge bases about various Burroughs systems products, and the sales personnel who use the automated configurator application to prepare complete and accurage configurations for customers. The same knowledge base used by the configurator application could potentially be used for diverse applications, including design revision, manufacturing scheduling, system pricing and maintenance. Though there may be little procedural commonality between the various applications, we gain reusability by consolidating common domain knowledge in an independent structure. While this is not software reuse in a strict sense, it is an effective reuse of knowledge that would more traditionally be embedded in application software (and hence rewritten anew for each application).

This separation also allows different domain experts to model individual parts of the system independently. Here the domain model turns out to share some of the advantages we associate with ASLs: because the domain model is defined in non-procedural terms, it is easier for the model to be independently maintained or created by application specialists who are not expert systems developers.

Just as many applications can use one knowledge base, an application can be written to work off multiple knowledge bases. For example, the functionality of the Burroughs configurator can be extended without modifying the application, by creating a model of a new system component. This is closer to our intuitive notion of software reuse, since the application can be adapted in a well managed way to different situations of use.

Extending Domain Analysis for Technology Selection

Our work in both program generation technology and knowledge-based systems has revealed a number of similarities in the domain analysis process for these respective areas, as well as similarities to domain analysis performed for the development of more conventional software parts technology as well (CAMP). This suggest that selection of appropriate technology for an application domain is best done in parallel with domain analysis, and that the domain analysis process

should be refined and extended to produce information relevant to this task.

A basic model for assessing the potential benefit of designing for reuse must provide a trade-off of the cost of the initial implementation, the projected number of future usages for the function, the average cost of each adaptation for reuse, and the cost of re-implementing rather than reusing for these instances. Domain analysis to support technology assessment must consider many additional factors. The domain analyst must look for commonality at different levels of abstractions and different phases of the software life cycle, and must look for common development activities and transformations as well as common static components. The following list is an initial set of questions that might be part of this process.

- (1) For a typical application program, what proportion of the processing consists of functions from the target domain? If applications tend to be predominantly invocations of doman functions, (e.g., database querying and reporting), an ASL might be appropriate. If domain functions are sparsely distributed, a library might be better. If the relative proportions of reuse ranges widely within the applications, a layered approach offering both direct interface to the library routines and an ASL shell may be indicated. (For example, most database systems provide both an embedded programmatic interface to database services and an independent query language, which may be interpreted or compiled.)
- (2) For a given category of reusable parts, how large would the necessary library of parts be? Would sophisticated cataloguing or pattern-matching tools be required to find the right routine in the library? If the size or complexity of the library passes a certain threshold, usage will drop because of retrieval effort. In this situation, it might be better to partition the library into smaller packages, or encapsulate some sets of routines with an automated part selection mechanism accessed by an ASL.
- (3) What is the expected distribution of usage along the various dimensions of

component variation (time/space, parameterization, binding mechanism)? For example, are components accessed as procedures, functions, tasks, or stand-alone programs, or a mix of these? If usage patterns are clustered along one axis, generation techniques may be appropriate. If usage needs vary widely, creation as needed and storage in the library might make most sense.

- (4) Are the parameter choices for a routine "flat" or "tree-structured"? A subroutine requires a fixed number of parameters (though defaults can be provided as in Ada). An ASL has more flexibility over parameter choices, but will still require the inputs in a batch mode. An expert system application could prompt intelligently and constrain choices further in the process as a result of previous decisions.
- (5) How deterministic are the functions common to the domain? Are they definable directly as functions in software, deterministic algorithms that can be incorporated in a generation system, or a set of rules, procedures and heuristics, for which an expert system might be an appropriate implementation?
- (6) How critical is the efficiency of the final code? If performance is not critical (such as in prototyping environment) conventional parts may be sufficient. If performance constraints are high, but parameterization does not vary widely, it may still be feasible to store discrete optimized parts, but more ancillary descriptions of optimization priorities and benchmarks will need to be maintained along with the software part. If both performance and flexibility are required, program generation techniques may be required to achieve adequate reuse.
- (7) How modifiable are the system requirements? Is the customer willing to change specifications to suit existing characteristics? If so, conventional reuse techniques will be more applicable.

This list is by no means a complete set of criteria for evaluation; nor are the

interpretations of the criteria iron-clad. The eventual goal would be a set of guidelines that a software manager could apply when considering the (re)automation of a specific domain, in order to choose the appropriate technology.

Hybridizations of technologies

Near-term (application-specific) technologies for software reuse, whether software libraries or ASLs, will cover only a small proportion of the large-scale, real-time applications of most concern, because these systems represent the intersection of multiple application domains (in the restricted sense described above), at disparate levels of formalization and standardization.

This does not mean that technologies for software reusability can have only an incremental impact on large system development in the near term. To achieve an impact on these systems adequate to the productivity goals of the STARS program, it is necessary to support a mix of horizontal and vertical domains; that is, both domains defined in terms of application areas in the real world (communications, air traffic control, etc.), and those that cut across traditional application boundaries, such as mathematical subroutines, manipulation of data structures, or support of software development activities. This strategy plays a key role in our plans to incorporate ASLs as an integral component of SDC's Common Software Environment (SDC-CSE) (SDC85). We plan to define ASLs tailored to several "axes" within the environment: (1) project roles associated with software life cycle phases (programmer, designer, requirements analyst) and skill levels: (2) architectural features of the environment (such as database interaction, project communication, or configuration management); and (3) additional ASLs supporting the specific application area of the project.

 ℓ_{33}

Because different domains are best suited for particular technical approaches, this mix of domains must be supported by promoting alternative technologies with the most potential for near-term cost-effectiveness, and developing techniques for the hybridization of these technologies wherever possible. For example, application generators have been most successfully used for areas like

database management, where typical programs do little but access the database and present the data. They are currently less suitable for domains where generated functionality is interspersed with arbitrary computation. Techniques for infiltrating code produced by application generators with handwritten code (or vice versa) would greatly expand the scope of use for these tools (Volkenburgh). Similarly, libraries of reusable software should be designed to accommodate the inclusion of hand-written components and automatically generated or transformed components in a uniform (and, to some degree, caller-transparent) manner.

The integration of specification and generation techniques with reusable software parts could facilitate effective reuse of these parts. When a software component library achieves a sufficient complexity one or more ASLs could be defined as a natural and efficient user interface to the library. The selection of software parts is automatically performed by the generation system, which does so on the basis of its built-in knowledge of the syntax and semantics of the software parts. Usages of the reusable software parts are linked by code automatically generated from the specification. This method guarantees correct and effective usage of the reusable parts. By allowing both access to the ASL interface and direct access to the underlying library of routines, maximum flexibility will be available when required; the ASL can be cleaner, since it will not have to accommodate as many pathological 'special cases'.

Finally, we see great potential for the application of knowledge-based techniques to parts composition, generation, catalogueing and tailoring systems. We believe the appropriateness of this technique will increase as more expertise is gained with conventional parts management systems technology.

General Issues

Advancing our understanding of appropriate matching of reuse technology to application domains is not going to solve all the difficult issues involved in reusing software. Designing for reuse is inherently more complex than writing special-purpose applications, because one sets out to solve a class of problems rather than one specific

problem. Thus, we should anticipate that each technology will present its own challenges in design. But also, certain problems that have been encountered in software components technology may reappear at a different level with application generators or expert systems. In the interest of a realistic perspective technologies to solve, we offer a few issues that appear common to all the approaches described above.

Application Specificity

Software reusability becomes more feasible, regardless of the technology involved, in direct proportion to the software's degree of specificity to a particular application domain. This is confirmed by the domains in which subroutine libraries have been most successful, such as libraries of mathematics, graphics, or operating system routines. The critical problems in library configuration management, cataloguing and retrieval quickly push the state of the art when the scope or complexity of the library gets too large. This is also a key to the strategy of very high-level application-specific languages in contrast to attempts to define general-purpose high-level specification languages. By confining language scope to small, clearly defined domains, it is possible for ASL processors to generate efficient code of · production quality. Finally, this observation is consistent with the general thrust in the expert systems area toward concentration on domain-specific expertise rather than general knowledge or problem-solving.

Separation of Volatile from Stable Information

One limiting factor to capturing commonality in a domain is the relative degree of volatility, or frequency of change, of the information in the domain. For example, one does not want to embed monthly pricing information in a program generation system that would have to be recompiled with each price shift. Though the rules used in knowledge-based systems might appear to support this sort of change better than a compilation system, it would appear that a knowledge-based application of any size and longevity also needs an auxiliary mechanism to handle rapidly changing knowledge. The

AI department at SDC has been involved with work on linking knowledge bases with loosely coupled conventional databases to achieve the necessary separation of volatile, time-dependent information from more stable domain-dependent knowledge. Viewed in this way, the database in effect functions as an extremely flexible and maintainable system for passing a large number of parameters to the system. A program generation system or a software parts composition system could make use of the same sort of facility.

Modularity

The need for modularity in large systems is not allayed by the introduction of ASLs, component libraries or expert systems. Instead, it reasserts itself at new levels of abstraction. Libraries should be partitioned into intuitively cohesive collections of services, modularized according to the same principles of good software engineering that are helping to make hand-written software tractable. (This conforms with the state of practice in the standard C libraries of UNIX*, or the intention of the package mechanism in Ada.)

We have advocated the creation of small narrowly defined ASLs rather than new large, general-purpose languages (since our purpose is not to reinvent Ada). In a environment where production of specialpurpose languages for software development has become economically justifiable, we must begin to modularize the languages in our environment with the same care that we create reusable subroutines. By keeping ASLs small, cohesive and single-function, we increase the ways in which these languages can be linked together to form new tools. We are currently investigating the theoretical problems in specifying shareable sublanguage ASLs that can be reused in different contexts. For example, an ASL for string pattern matching might be used within many other ASLs. We should be able to define it as a separate language and invoke it as such from other language specifications. Finally, modularity in knowledge bases is a key to the tractability of large expert systems, as we have seen.

One implication of this recurring modularity is that components management will be needed at all levels in a hybrid technology environment. ASLs will need to be maintained, catalogued and reused just as we currently propose for subroutines. Further down the pike, knowledge bases themselves might reside in libraries as well.

Maturity of Domain Knowledge

The technology suitable to an application domain depends closely on the relative maturity and stability of the domain, and the presence of a firm basis for standardization and the consolidation of expertise. This is borne out of examples such as the widespread use of application generators for well-understood domains such as business software, and the successful libraries of standard routines. This implies that efforts to introduce software libraries or application generators in highly unstable or innovationintensive software development environments may constitute a premature introduction of reuse technology. It may result in simple wasted effort or premature, hence ineffectual standardization. Instead, we advocate the incremental and evolutionary approach of initially tackling narrowly defined, highly constrained and wellunderstood sub-domains within such application areas. In this phase, library support or ASL support might be equally feasible depending on the profile of the domain. As our knowledge of an application domain matures, we will evolve naturally through a progression of technologies to support reuse, beginning with ad-hoc reuse, continuing through development of standard libraries of routines for common functions, then automating the composition of these functions through higher-level application generators, to eventual knowledge-based support. This corresponds with the evolution in database technology, which may serve as the classic example (to date) of a reuse-intensive software domain.

Conclusion

The taxonomy of reusability technologies and criteria for domain presented here are initial suggestions. Much work needs to be done to make this framework into a comprehensive methodology that can be of general use within the indistry, though there is already a large body of experience to guide

this work. This knowledge should be consolidated and codified, through industry-wide forums for discussion such as the STARS Applications Area Workshop. Once some consensus has been reached on the dependability of these criteria, the proposed STARS Reusability Guidebook would be an excellent avenue for making these guidelines available to the software industry as a whole.

*UNIX is a registered trademark of AT&T Bell Laboratories.

References

(Batz83) Batz, J.C., Cohen, P.M., Redwine, S.T., Rice, J.R., "The Application-Specific Task Area", IEEE Computer, 16:11, pp. 78-85, November 1983.

(CAMP) Anderson, C.M., McNicholl, D.G., "Contract FO 8635-84-C-0280, Common Ada Missile Parkages (CAMP):Preliminary Technical Report, Vol. 1", in STARS Workshop Proceedings, April 1985.

(Cheatham) Cheatham, T.E., "Reusability through Program Transformations", IEEE Trans. on Software Engineering, Special Issue on Software Reusability, SE-10:5, September 1984.

(Freeman83) Freeman, M.W., Hirschman, L., McKay, D.P., Miller, F.L., Sidhu, D.P., "Logic Programming Applied to Knowledge-Based Systems, Modelling, and Simulation", Proceedings of the Conference on Artificial Intelligence, Oakland University, April 1983, pp. 177-193.

(Freeman85) Freeman, M.W., "Case Study of the BEACON Project: The Burroughs Browser/Editor and Automated Configurator", IEEE Televido Symposium on Expert Systems in Prolog, Dec. 9, 1985.

(Horowitz) Horowitz, E., Munson, J.B., "An Expansive View of Reusable Software", IEEE Trans. on Software Engineering, Special Issue on Software Reusability, SE-10L5, September 1984, pp. 477-487.

(IEEE) IEEE Trans on Software Engineering, Special Issue on Software Reusability, SE-10:5, September 1984.

(Kastens) Kastens, U., "Ordered Attribute Grammars", Acta Informatica, Vol. 13, 1980, pp. 229-256.

(Knuth) Knuth, D., "Semantics of Context-Free Languages", Math. Systems Theory, 5:1, 1971, pp. 127-145.

(McIlroy) McIlroy, M.D., "Mass-produced Software Components", Software Engineering Concepts and Techniques, 1968 NATO Conf. Software Eng., J.M. Buxton, P. Naur, and B. Randell, Eds. 1976, pp. 88-98.

(SDC85) T. Payton et. al., Architectural Description of the SDC Common Software Environment (SDC-CSE), Under contract to Naval Air Development Center, Warminster, PA. (CDRL A002, Contract No. N62269-85-C-0485) January, 1986.

(Standish) Standish, T.A., "An Essay on Software Reuse", IEEE Trans. on Software Engineering, Special Issue on Software Reusability, SE-10:5, September 1984, pp. 494-497.

(Teitelbaum81) Teitelbaum, T., Reps, T., "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", CACM, 24:9, 1981, pp. 563-573.

(Volkenburgh) Volkenburgh, G.V., "Information Package for Workshop on Reusable Components of Application Software", STARS Workshop Proceedings, April 1985.

Creating Reusable Ada® Software

Ed Berard

®Ada is a registered trademark of the U.S. Government (Ada Joint Program Office)

©1986 EVB Software Engineering, Inc.



The primary objective of this set of notes is to make the audience aware of some of the more important issues relating to the creation of reusable Ada software. Specifically, these notes are designed to touch upon the "nuts and bolts" issues. Since the time allotted for presentation is less than one day, the material is intentionally brief. It is assumed that the audience has at least a reading knowledge of the Ada programming language, and has developed at least one piece of serious software.

Some of the concepts contained in these notes were originally developed by Grady Booch (Rational, Inc.), and will be amplified in his soon to be published book (Software Components With Ada, Benjamin/Cummings). Specifically, the terminology associated with reusable modules and the concept of subsystems were first described by Mr. Booch in previous tutorials. While there is no formal working arrangement between Mr. Booch and EVB Software Engineering, Inc., EVB recognizes and appreciates the pioneering work done by Mr. Booch.

©1986 E.B Software Engineering, Inc.



 \mathfrak{M}

If Hardware People Thought Like Software People

- "There are some unused 'op codes' in this CPU for this specific application. Why don't we remove the extra ones?"
- "There are 613 unused bytes of RAM for this application. Let's redesign the hardware so that we can remove the extra memory?"
- "Using an 'off-the-shelf' CPU is for wimps. Let's design our own application-specific CPU for this application. The same goes for integrated circuits in general."

©1986 EVB Software Engineering, Inc.

- Reusability: the extent to which a module can be used in multiple applications. (This definition skirts the issue of how much change, if any, might be required in the module's code.)
- Portability: The ease with which software can be transferred from one computer system or environment to another.
- Modifiability: The ease with which a piece of software may be changed to suit a specific application.

(Continued)

- Maintainability: The ease with which maintenance of a functional unit can be performed in accordance with prescribed requirements.
- Reliability: The probability that software will not cause the failure of a system for a specified time under specified conditions.
- Abstraction: A view of a problem that extracts the essential information relevant to a particular purpose and ignores the remainder of the information.

(Continued)

- Functional Abstraction: A view of a problem that permits the user to know precisely about the input-output specification while hiding the underlying implementation of the function itself. (This permits reusability of the function for varying data of a fixed type.)
- Data Abstraction: A view of a problem that hides both the underlying structure of the input-output data and the underlying functionality of a module. The user may occasionally know some of the details of the underlying algorithms used in the module.

©1986 EVB Software Engineering, Inc.

(Continued)

- Process Abstraction:
 Similar to data abstraction, but differs in having an independent executing thread of control that determines the order in which operations become available for execution (includes concurrent processes).
- Usability: The ease with which a piece of software may be used for a specific application.

Reusability Axioms

- Reusability is not an absolute (or discrete) concept.
- The Ada programming language provides reusability concepts which are fundamentally different from those in most other commonly used programming languages.
- Reusability is increased when software engineers achieve the goals of software engineering by adhering to the principles of software engineering.
- Management must encourage the reuse of software, and software engineers must both design and use reusable software.

©1986 EVB Software Engineering, Inc.

Reusability Axioms

(Continued)

- Reusable software must be promulgated within an organization.
- Reusability must be defined, measured, recorded, and increased.
- Software engineers must avoid language/implementation tricks.
- Software engineers must know what factors affect reusability.
- Software engineers must know what factors affect portability.
- Reusability and portability are enhanced when modules are functionally cohesive and loosely coupled, i.e., they are highly independent.

Reusability Axioms

(Continued)

- Reusability and portability are enhanced when modules have well-defined interfaces.
- Software engineers must know what is general and what is specific to an application.
- Robust modules (created through defensive programming) are more reusable than non-robust modules.
- Practice conceptual integrity.
- There are times when reusability is not important.

What Can Be Reused?

- Code Fragments
- Modules (components)
- Subsystems (Rational/Booch definition)
- Tools
- Designs

General Ada Coding Style Guidelines for Ada Reusability

- Use meaningful identifiers.
- Make frequent use of attributes.
- Avoid literal constants.
- Use named parameter association.
- Avoid the "use" clause.
- Use the "renames" only to expose part of an abstraction.
- Use fully-qualified names
- Create adequate, concise, and precise comments.

Style Guidelines

(Continued)

- Fully exploit the separate compilation features of the Ada language.
- Make frequent use of subunits.
- Avoid default values for descriminants, record field values, and formal parameters.
- Avoid pragmas.

- Avoid "unchecked deallocation."
- Avoid "unchecked_conversion."
- Avoid anonymous types.
- Avoid pre-defined and implementation-defined types.
- Avoid optional language features.

Style Guidelines

(Continued)

- Avoid attention to underlying implementations.
- Avoid restrictive modules.
- Strive for limited private types.
- Make frequent and appropriate use of packages.
- Make very frequent and appropriate use of generics.
- Isolate, and clearly identify environmentally-dependent code.
- Watch out for assumptions about garbage collection.



Reusable Modules

Let us consider the implemention of a data structure in the Ada language. For purposes of example, consider a stack. A stack is a list to which we may add or delete items from one end only, i.e., it is a last-in-first-out data structure. The question is: "how are we to implement a stack in the Ada language?"

Reusable Modules

(Continued)

- FORTRAN Mindset:
 Implement the stack as an array
- Pascal/C Mindset: Implement the stack as a linked list
- Primitive Ada Mindset: Implement the stack as a package
- Adequate Ada Mindset: Implement the stack as a generic package
- Advanced Ada Mindset: Implement the stack as a family of generic packages.

Reusable Modules

(Continued)

The experienced software engineer recognizes that the time/space behavior of a component is as important as its functional behavior. (This emphasis on functional behavior is often the result of a functional decompostion approach to the design of software.) When we speak of time behavior, we are concerned with the behavior of the component in a concurrent environment. When we speak of space behavior, we are concerned with how a component utilizes memory.

Characteristics of Highly-Reusable Operations

- Primitive: The operation cannot be implemented without knowledge of the underlying implementation of the object
- Complete: We have a minimal set of primitive operations which will allow us to implement all necessary operations for the object
- Sufficient: We have added additional operations to our minimal set of operations to enhance the usability of our abstraction.

Classification of Objects

- Monolithic: The object is not composed of substructures, e.g., stacks and queues
- Polylithic: The object may be viewed as being composed of identical substructures, i.e., the object is recursively defined, e.g., lists and trees

Classification of Operations

- Selector: Returns information about an object, but cannot change the state of the object
- Constructor: Changes the state of an object, often does not return information about the object
- Iterator: For objects which have a structure, allows us to visit each node of the structure and to perform some operation at each node. This operation is characteristically a selector operation.

Taxonomy of Primitive Reusable Modules

- Bounded/Unbounded
- Iterator/Non-Iterator
- Managed/Unmanaged
- Concurrent/Sequential/Guarded/ Controlled/Multiple
- Priority/Non-Priority
- Balking/Non-Balking
- Limited/Non-Limited



Bounded Vs. Unbounded

- Bounded: There is a specified upper limit to the number of nodes in the data structure, which is specified at declaration time. (Note: The underlying implementation is accomplished via sequential allocation, and the use of any dynamic variables is strongly discouraged.)
- Unbounded: The data structure is free to grow or shrink based on available computer resources. These are implemented using linked allocation.

Iterator Vs. Non-Iterator

- Iterator: The component provides an iterator operation, i.e., a means of visiting all the nodes in the underlying abstraction and performing some action at each node.
- Non-Iterator: The component does not provide an iterator capability

Managed Vs. Unmanaged

- Managed: The component provides its own memory management, e.g., it maintains a "free list" of available nodes rather than depending on features such as unchecked_deallocation
- Unmanaged: the component provides no memory management capabilities, i.e., it depends on those provided by the environment

- Sequential: The component will behave as expected in a non-concurrent environment. In a concurrent environment, the component may be subject to data and process pollution.
- Concurrent: The component will behave in a reasonable manner in a concurrent environment, i.e., the component is constructed so as to avoid data and process pollution. No user action is required

©1986 EVB Software Engineering, Inc.

'WV

(Continued)

orded: The component provides the user with the capability of using the component in a concurrent environment, i.e., a semaphore mechanism to "lock" and "unlock" objects. While this is very dangerous (as opposed to concurrent components) the user has the ability to easily construct higher-level operations from the "atomic" operations provided in the guarded component.

(Continued)

• Controlled: The user of the component will prevent the object from being simultaneously accessed by two or more processes. The component, in turn, will protect any state information (e.g., a free list) contained within the component. Note that concurrent components may be built on top of controlled components.

(Continued)

Multiple: The component provides for multiple reads (selector operations) of an object while sequentializing writes (constructor operations) to the object. This allows for a high degree of concurrent access to an object while preventing corruption of the object or state information associated with the object.

Priority Vs. Non-Priority

- Priority: The nodes in the data structure are ordered based on a priority scheme, e.g., a priority queue. (Note: In a priority structure, operations on the nodes are dependent on both those normally associated with the abstraction, and on the priority of the items placed in the structure.)
- Non-Priority: The items in the data structure are not treated on any priority basis.

Balking Vs. Non-Balking

- Balking: Items may be removed from a data structure in a manner other than that normally associated with the abstraction, e.g., in a balking queue, items may be removed without first bringing them to the front of the queue.
- Non-Balking: The component provides no other operations for the removal of items from a data structure other than those normally associated with the abstraction

Limited Vs. Non-Limited

- Limited: The abstraction is a very large data structure with specific bounds, however, the underlying implementation is accomplished via linked allocation, e.g., sparse matrices
- Non-Limited: The underlying implementation is consistent with the abstraction, i.e., bounded components are implemented using sequential allocation and unbounded components are implemented using linked allocation.

Concurrency Issues

- Data and process pollution
- Indeterminacy
- Deadlocking
- Friendly vs. unfriendly tasking implementations
- Degree of concurrency
- Guarded vs. Concurrent components



Garbage Collection Issues

- Use or non-use of access types
- Allocation of heap storage, e.g.:

for Item'Storage_Size use
5*Kilo Bytes;

- Use of "unchecked deallocation"
- Time to allocate and deallocate heap storage

Compiler Issues

- Avoidance of compiler dependent features
- Considering compiler optimization features
- Avoidance of "tuning" the Ada code to any specific compiler (or hardware)

Exceptions In Reusable Components

E CENT

(

- Use of exceptions to report exceptional conditions
- Designing and exporting wellnamed exceptions
- Noting the use of exceptions in the comments for program units
- Handling exceptions in Ada tasks

Efficiency Issues

- Inverse relationship between efficiency and reliability
- Efficiency vs. reusability
- Efficiency vs. portability
- Use of pre-existing, proven algorithms
- How much efficiency should you strive for
- Exporting objects vs. exporting types (this is also a strong usability issue)

Subsystems

Subsystems are collections of packages (mostly generic packages) which behave logically like packages, i.e.:

- The collection is treated as a unit (even though the syntax and semantics of the Ada language may not necessarily be used to enforce this)
- Objects and types, as well as operations may be exported
- Some of the operations, objects, and types are visible while others are hidden

Subsystems

(Continued)

- The hardware analog of a subsystem is a populated printed circuit (pc) board
- Subsystems are of a higher level of abstraction than packages
- Like populated pc boards, subsystems are not stand-alone applications, but are used to construct applications
- Examples of subsystems include menu subsystems and windowing subsystems
- Subsystems are less common than reusable modules and tend to be more vertical than horizontal in their application areas

Tools are stand-alone applications. The usual connotation is that they are used by software engineers to automate various parts of the software life-cycle. Booch classifies tools as:

- Utilities
- Filters (Kernighan and Plauger)
- Sorting
- Searching

(Continued)

Often tools are considered within the context of a software engineering environment. This implies that their reusability will be strongly connected to their ability to interact /integrate with the environment, and their ability to interact/integrate with other tools. In the Ada world, this requires some additional concepts:

- DIANA
- Discrete tools (e.g., tools in a UNIXTM or VMS environment)
- Diffuse tools (e.g., tools in the Rational Environment)

(Continued)

Tools may be either stand-alone or invokable from within an application. For example, a tool which analyzes the complexity of Ada source code might be parameterized so that it may be called from within an application and pass the complexity information to the calling unit as an abstraction. Note that this increases the reusability of the tool.

(Continued)

To increase the reusability and portability of a tool, the software engineer should:

- Isolate and clearly identify any implementation-dependent modules
- Use packages instead of files for I/O.
- Follow the previously mentioned Ada coding style guidelines
- Not attempt to tailor the tool for any specific Ada implementation

Reusable Designs

- With all due respect to Ada as a DL, we will define a software design as any non-code software, e.g., documentation.
- A previously-existing design for a non-Ada implementation could be reused to implement the application in the Ada language.
- With a well-engineered design (most likely not produced via a functional decomposition approach), parts of the design might be incorporated into (reused) another Ada application.

Designing With Reusability In Mind

- Two basic problems: produce reusable software as a by-product of the design effort, and to make use of previously existing reusable software.
- Some software development methodologies are more prone than others to produce reusable software.
- Reusable software can be used in both a top-down and a bottom-up design effort.
- Rapid prototyping is possible with carefully constructed reusable components.

Designing With Reusability In Mind

(Continued)

One of the largest impediments to the creation and use of reusable software is the creation of a hardware architecture first, and then requiring that the software be designed around the hardware. A far better approach is software first, i.e., the software is designed first, and then the hardware is designed and built to support the software.

Major Obstacles to Reusable Software

- NIH
- "Only wimps use someone else's software."
- Contracting procedures which encourage "re-invention of the wheel," large staffing, and low quality software
- Lack of confidence in the quality of potentially reusable software, i.e., lack of a formal certification mechanism
- Lack of technical expertise
- Unawareness of technology, or reusable components
- Lack of useful tools

An Example of Ada Code

1 Defining the Problem

1.1 Stating the Problem

Create a generic bounded stack package.

1.2 Analysis and Clarifications of the Givens

- 1. The following is a definition of a linear list: "A linear list is a set of n >= 0 nodes X[1], X[2], ..., X[n] whose structural properties essentially involve only the linear (one-dimensional) relative positions of the nodes: the facts that, if n >= 0, X[1] is the first node; when 1 < k < n, the k th node X[k] is preceded by X[k-1] and followed by X[k+1]; and X[n] is the last node." (See [Knuth, 1973].)
- 2. A stack is defined to be "a linear list for which all insertions and deletions (and usually all accesses) are made at one end of the list" ([Knuth, 1973]). This end is usually called the *top* of the stack. Notice that the above definition *implicitly* states the Last-In-First-Out (LIFO) property of a stack.
- 3. By bounded, we mean that the *maximum length* of a given stack does not change. Thus, a user of this generic must specify the desired length of a stack when the stack object is declared.
- 4. The stack abstraction must be sufficient, complete, and primitive. Sufficient means that a sufficient variety of operations are provided to allow the user of the abstraction to implement what is needed. Complete means that all aspects of the abstract behaviour of the abstraction are captured. Primitive means that only those operations that could not be implemented effectively without access to the underlying implementation are provided.
- 5. The following operations (meeting the tradeoffs of the criteria defined above) are defined inside the stack package: push down an element onto a stack, pop up an element off a stack, check whether a stack is empty or full, find out how many elements are currently in a given stack, find out the top element in the stack, peek at the n th element below the top (where l <= n <= current number of elements in the stack), clear a given stack (i.e., create an empty stack), test two stacks for equality, and copy one stack to another.
- 6. If a given stack is *empty* and the user tries to *pop* an element off the stack, an exception (Underflow) will be raised.
- 7. Overflow will be raised if a user tries to push an element onto a full stack.
- 8. Element Not Found will be raised if a user tries to peek at a non-existent element or tries to find out the top element in an empty stack.
- 9. We are not concerned with what types of elements are put on the stack. As many kinds of elements as possible should be allowed.

2 Developing an Informal Strategy

A user will be able to push an element onto a stack, pop an element off a stack, find out whether a stack is empty, find out whether a stack is full, determine the number of elements in a stack, find out the top element in a stack, peek at an element in a stack, and clear a stack. The user will also be provided with a means to test two stacks for equality and a means of copying the contents of one stack to another.

3 Formalizing the Strategy

3.1 Identifying the Objects of Interest

3.1.1 Identifying Objects and Types

A <u>user</u> will be able to push an <u>element</u> onto a <u>stack</u>, pop an <u>element</u> off a <u>stack</u>, find out whether a <u>stack</u> is empty, find out whether a <u>stack</u> is full, determine the number of <u>elements</u> in a <u>stack</u>, find out the <u>top element</u> in a <u>stack</u>, peek at an <u>element</u> in a <u>stack</u>, and clear a <u>stack</u>. The <u>user</u> will also be provided with a means to test two <u>stacks</u> for equality and a means of copying the contents of one <u>stack</u> to another.

Objects	Space	Identifier
user	Problem .	
element	Solution	Element
stack	Solution	Stack
top element	Solution	(=Element)

()(a

3.1.2 Associating Attributes with the Objects and Types of Interest

Element

- is an abstract type
- can be copied
- can be tested to see if it is equal to another element

Stack

- defines an abstract type
- its fixed maximum length does not change
- can be empty
- can be full
- can contain up to some user-defined maximum number of items
- elements in a stack are pushed on to the top of the stack, or popped from the top of the stack, but all insertions and deletions are from that end only.
- The Last element In is the First element Out (LIFO)

3.2 Identifying the Operations of Interest

3.2.1 Identifying Operations

A user will be able to push an element onto a stack, pop an element off a stack, find out whether a stack is empty, find out whether a stack is full, determine the number of elements in a stack, find out the top element in a stack, peek at an element in a stack, and clear a stack. The user will also be provided with a means to test two stacks for equality and a means of copying the contents of one stack to another.

Operations	Space	Objects	Identifier
will be able	Problem		
push an element onto a stack	Solution	Stack	Push
pop an element off a stack	Solution	Stack	Pop
find out whether is empty	Solution	Stack	Is_Empty
find out whether is full	Solution	Stack	Is_Full
determine number of elements	Solution	Stack	Number_Of_Elements
find out the top element in a stack	Solution	Stack	Top_Of
peek at an element in a stck	Solution	Stack	Peek
clear a stack	Solution	Stack	Clear
will also be provided	Problem		
to test two stacks for equality	Solution	Stack	"="
copying the contents of one stack to another	Solution	Stack	Сору



3.2.2 Associating Attributes with the Operations of Interest

Push

- Is a constructor
- puts a given element onto the top of a given stack
- raises an exception (Overflow) if a user tries to push an element onto a full stack

Pop

- Is a constructor
- pops the top element off a given stack
- raises an exception (Underflow) if a user tries to pop an element off an empty stack

Is_Empty

- Is a selector
- has a true value if the given stack is empty; false otherwise

Is_Full

- -- Is a selector
- has a true value if the given stack is full; false otherwise

Number_Of_Elements

- Is a selector
- is the current number of elements in a given stack

Top Of

- Is a selector
- -- allows a user to look at the content of the top of a given stack, i.e., the top element in a given stack
- -- raises an exception (Element_Not_Found) if the stack is empty

Peck

- Is a selector



- raises an exception (Element_Not_Found) if a user tries to peek at a non-existent element (e.g., there are currently 5 elements in a stack and a user tries to peek at the 8th element)

Clear

- Is a constructor
- initialize stack to empty stack

"ב"

- -- Is a selector
- returns true if two stacks are equal
- -- two stacks are equal if the following conditions are all true:
 - 1. the current number of elements in both stacks are equal, and
 - 2. the values of the corresponding elements in both stacks are equal

Copy

- -- Is a constructor
- -- copy one entire stack to another
- -- will raise an exception (Overflow) if the destination stack has an upper bound that is smaller than the number of elements in the source stack

3.2.3 Grouping Operations, Objects, and Types

Stack

Push

Pop

Is_Empty

Is_Full

Number_Of_Elements

Top_Of

Peck

Clear

"_"

Copy

Elements

«none» (is part of the interface data to the other program units. See [EVB, 1985] page 3-38 # 3.)



3.3 Defining the Interfaces

3.3.1 Formal Description of the Visible Interfaces

The generic package Bounded_Stack contains:

```
the generic formal parameter: Element,
the type Stack,
and the following operations:

Push
Pop
```

Pop
Is_Empty
Is_Full
Number_Of_Elements
Top_Of
Peek
Clear
"_"
Copy,

and the following exceptions:

Underflow Overflow Element_Not_Found

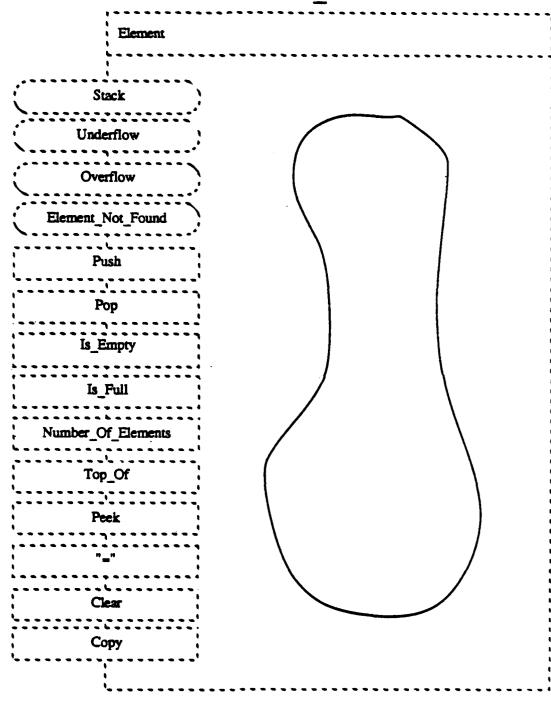


3.3.2 Analysis and Clarifications of High-Level Design Decisions

1. The generic formal type parameter Element will be of type private to satisfy the criteria set down in the Analysis and Clarifications fo the Givens (1.2) number 9 and the fact that assignment of Elements will be needed in order to put things onto the stack.



3.3.3 Graphic Annotation of the Visible Interfaces Bounded_Stack





3.4 Implementing the Solution

3.4.1 Implementing the Operation Interfaces

```
generic
```

```
type Element is private ;
    - Element is the type of object that will be put on the stack
package Bounded_Stack is
   -- This generic bounded stack package provides stack manipulation
   -- operations neccessary for most applications where stack data
   -- structure is needed. In building this package, we were striving
   -- for the following: operations provided in this package must
   -- be primitive and complete and sufficient.
   -- A primitive operation is an operation which can NOT be
   -- implemented effectively WITHOUT knowing the underlying
   -- representation of a particular object (in this case our
   -- object is STACK). By complete we mean that a user will find
   -- that the following operations will be the ONLY operations
   -- needed to manipulate a bounded stack in almost any application.
   - Whenever neccessary, a user will be able to build other
   -- operations based only on the operations provided in this package.
   -- Written by Johan Margono, reviewed by (in alphabetical order):
   -- B.D. Balfour, E.V. Berard, G.E. Russell
   - Author
                   Revision
                              Date
                                           Reason
   -- J. Margono
                   1.0
                              9 Jul 1985
   - J. Margono
                   2.0
                             19 Mar 1986
                                            "type Stack (Length : ... " is
                                           changed to "type Stack
                                            (Maximum_Length : ... " to be
                                           consistent with our naming
                                           convention
    - (c) 1986 EVB Software Engineering, Inc.
  type Stack (Maximum_Length : Positive) is limited private ;
  procedure Push (This_Element : in
                    Into_This_Stack : in out Stack) ;
     -- push This Element onto the top of Into This Stack;
     -- exception Overflow will be raised if Into_This_Stack is full
```

```
procedure Pop (Top Element
                              :
                                    out Element :
                Off This Stack : in out Stack) ;
   -- pop Top Element off Off This Stack;
   -- exception Underflow will be raised if Off This Stack is empty
function Is_Empty (This_Stack : in Stack) return Boolean ;
   -- returns true if This_Stack is Eepty; false otherwise
function Is_Full (This_Stack : in Stack) return Boolean ;
   -- returns true if This Stack is full; false otherwise
function Number Of Elements (In This Stack : in Stack) return Natural ;
   -- returns the current number of elements in In_This_Stack;
   -- eero is returned if In This Stack is empty
function Top Of (This Stack : in Stack) return Element ;
   -- returns the top element in This_Stack;
   -- exception Element Not Found is raised if This Stack is empty
function Peek (At Element
                              : in Natural ;
                In This Stack : in Stack) return Element ;
   -- returns the nth element in In This Stack 1 <= n <= length
   -- (n is equal to At Element). If At Element is greater than the
   -- current number of elements in In This Stack, the exception
   -- Element Not Found will be raised (note : Element Not Found will
   -- also be raised if In_This_Stack is empty)
procedure Clear (This_Stack : in out Stack) ;
   -- makes This_Stack an empty stack
function "=" (Left : in Stack ; Right : in Stack) return Boolean ;
   -- returns true if :
        1. number of elements in Left is equal to number of elements
           in Right, AND
        2. values of elements in Left is equal to values of elements
           in Right (i.e., in corresponding slots);
   -- false otherwise
```

T.

```
procedure Copy (This_Stack : in
                                      Stack ;
                 Into
                                  out Stack) ;
   -- copy the contents of one stack into another stack (Into);
   -- will raise Overflow if the destination stack (Into) has
   -- length that is smaller than the number of elements in the
   -- source stack (This_Stack)
Underflow
                    : exception ;
   -- raised if a user tries to pop an element off an empty stack
Overflow
                    : exception ;
   -- raised if a user tries to push an element onto a full stack
Element_Not_Found : exception ;
   -- raised if a user tries to find the top of an empty stack
  -- or Peek at an empty stack or peek at non-existent element
```

```
Empty_Stack_Index : constant := 0 ;

type Contents_Of_Stack is array (Positive range <>) of Element ;

type Stack (Maximum_Length : Positive) is record
    Top : Natural := Empty_Stack_Index ;
    Contents : Contents_Of_Stack (1 .. Maximum_Length) ;
end record ;
-- Stack is initially Empty (i.e., Top = Empty_Stack_Index)

end Bounded_Stack ;
```

3.4.2 Stepwise Refinement of the Highest-Level Program Unit

package body Bounded_Stack is -- This generic bounded stack package provides stack manipulation -- operations neccessary for most applications where stack data -- structure is needed. In building this package, we were striving -- for the following: operations provided in this package must -- be primitive and complete and sufficient. -- A primitive operation is an operation which can NOT be -- implemented effectively WITHOUT knowing the underlying -- representation of a particular object (in this case our -- object is STACK). By complete we mean that a user will find -- that the following operations will be the ONLY operations -- needed to manipulate a bounded stack in almost any application. -- Whenever neccessary, a user will be able to build other -- operations based only on the operations provided in this package. -- Written by Johan Margono, reviewed by (in alphabetical order): -- B.D. Balfour, E.V. Berard, G.E. Russell -- Author Revision Reason 8 Jul 1985 -- J. Margono -- (c) 1986 EVB Software Engineering, Inc. procedure Push (This_Element : in Element : Into This Stack : in out Stack) is separate ; push This_Element onto the top of Into_This_Stack; -- exception Overflow will be raised if Into This Stack is full out Element ; procedure Pop (Top_Element : Off_This_Stack : in out Stack) is separate ; -- pop Top Element off Off This_Stack; -- exception Underflow will be raised if Off_This_Stack is empty function Is_Empty (This_Stack : in Stack) return Boolean is separate ; -- returns true if This_Stack is empty; false otherwise

©1986 EVB Software Engineering, Inc.

function Is_Full (This_Stack : in Stack) return Boolean is separate ;

-- returns true if This_Stack is full; false otherwise

```
function Number_Of_Elements (In_This_Stack : in Stack)
             return Natural is separate ;
   -- returns the current number of elements in In_This_Stack;
   -- zero is returned if In_This_Stack is empty
function Top_Of (This_Stack : in Stack) return Element is separate ;
   -- returns the top element in This_Stack;
   -- exception Element Not Found is raised if This Stack is empty
function Peek (At_Element
                            : in Natural ;
                In_This_Stack : in Stack) return Element is separate;
   -- returns the nth element below the top element in In_This_Stack
  -- (n is equal to At_Element). If At_Element is greater than the
  -- current number of elements in In This Stack minus one, exception
  -- Element Not Found will be raised (note : Element Not Found will
   -- also be raised if In This Stack is empty)
procedure Clear (This_Stack : in out Stack) is separate ;
   -- set This_Stack to be the same as an empty stack
```

```
function "=" (Left : in Stack ; Right : in Stack) return Boolean is
   -- returns true if :
       1. number of elements in Left is equal to number of elements
           in Right, AND
       2. values of elements in Left is equal to values of elements
           in Right (i.e., in corresponding slots);
   -- false otherwise
   -- NOTE: This function is included in the body of the package (as
           opposed to being implemented as a subunit) because,
            according to section 10.1, paragraph 3 of the Ada
            Language Reference Manual, "The designator of a
            separately compiled subprogram (whether a library unit
            or a subunit) must be an identifier."
   -- Written by Johan Margono, reviewed by (in alphabetical order):
   -- B.D. Balfour, E.V. Berard, G.E. Russell
                    Revision
   - Author
                                     Date
                                                   Reason
                                     8 Jul 1985
   - J. Margono
                    1.0
                                    20 Feb 1986
   -- B. Balfour
                    2.0
                                                   removed restriction that
                                                   stacks must have same
                                                   length (bounds)
     (c) 1986 EVB Software Engineering, Inc.
begin -- "="
   return Left.Contents(1 .. Left.Top) = Right.Contents(1 .. Right.Top) ;
end "=" ;
```

```
separate (Bounded_Stack)
procedure Push (This Element
                                : in
                                           Element ;
                 Into This Stack : in out Stack) is
   -- push This_Element onto the top of Into_This_Stack;
   -- exception Overflow will be raised if Into This Stack is full
   -- Written by Johan Margono, reviewed by (in alphabetical order):
   -- B.D. Balfour, E.V. Berard, G.E. Russell
   -- Author
                    Revision
                                                        Reason
    - J. Margono
                    1.0
                                     8 Jul 1985
   -- (c) 1986 EVB Software Engineering, Inc.
begin -- Push
   if Into_This_Stack.Tcp = Into_This_Stack.Maximum_Length then
      raise Overflow;
   else
      Into_This_Stack.Top := Into_This_Stack.Top + 1 ;
      Into_This_Stack.Contents(Into_This_Stack.Top) := This_Element ;
   end if;
end Push ;
```

```
E.
```

```
separate (Bounded_Stack)
procedure Pop (Top_Element
                                   out Element ;
                Off_This_Stack : in out Stack) is
    - pop Top_Element off Off_This_Stack;
      exception Underflow will be raised if Off_This_Stack is empty
   -- Written by Johan Margono, reviewed by (in alphabetical order):
    -- B.D. Balfour, E.V. Berard, G.E. Russell
    - Author
                    Revision
                                     Date
                                                        Reason
                                     8 Jul 1985
    - J. Margono
                    1.0
     (c) 1986 EVB Software Engineering, Inc.
begin -- Pop
   if Off_This_Stack.Top = Empty_Stack_Index then
      raise Underflow ;
      Top_Element := Top_Of (This_Stack => Off_This_Stack) ;
      Off This Stack.Top := Off This Stack.Top - 1;
   end if;
end Pop ;
```

```
separate (Bounded_Stack)
function Is_Empty (This_Stack : in Stack) return Boolean is
     returns true if This_Stack is empty; false otherwise
    - Written by Johan Margono, reviewed by (in alphabetical order):
   -- B.D. Balfour, E.V. Berard, G.E. Russell
    - Author
                    Revision
                                    Date
                                                        Reason
    - J. Margono
                    1.0
                                     8 Jul 1985
   -- (c) 1986 EVB Software Engineering, Inc.
begin -- Is_Empty
   return This_Stack.Top = Empty_Stack_Index ;
end Is_Empty ;
```

```
separate (Bounded_Stack)
function Is_Full (This_Stack : in Stack) return Boolean is
    - returns true if This_Stack is full; false otherwise
   -- Written by Johan Margono, reviewed by (in alphabetical order):
   -- B.D. Balfour, E.V. Berard, G.E. Russell
    - Author
                    Revision
                                     Date
                                                        Reason
    - J. Margono
                                     8 Jul 1985
   -- (c) 1986 EVB Software Engineering, Inc.
begin -- Is_Full
   -- check if top of stack is equal to length of stack
   return This_Stack.Top = This_Stack.Maximum_Length ;
end Is_Full ;
```

```
separate (Bounded Stack)
function Number_Of_Elements (In_This_Stack : in Stack) return Natural is
    -- returns the current number of elements in In_This_Stack;
   -- zero is returned if In_This_Stack is empty
   -- Written by Johan Margono, reviewed by (in alphabetical order):
   -- B.D. Balfour, E.V. Berard, G.E. Russell
   -- Author
                   Revision
                                                        Reason
   -- J. Margono
                   1.0
                                     8 Jul 1985
   -- (c) 1986 EVB Software Engineering, Inc.
begin -- Number_Of_Elements
   return In_This_Stack.Top ;
end Number_Of_Elements ;
```

```
separate (Bounded Stack)
function Top_Of (This_Stack : in Stack) return Element is
    -- returns the top element in This_Stack;
   -- exception Element_Not_Found is raised if This_Stack is empty
    -- Written by Johan Margono, reviewed by (in alphabetical order):
   -- B.D. Balfour, E.V. Berard, G.E. Russell
    -- Author
                    Revision
                               Date
                                            Reason
                               8 Jul 1985
    - J. Margono
                    1.0
   -- J. Margono
                    2.0
                              19 Mar 1985
                                            Constraint_Error will not
                                            be raised if pragma SUPPRESS
                                            is used
   -- (c) 1986 EVB Software Engineering, Inc.
begin -- Top_Of
   if This_Stack.Top = Empty_Stack_Index then
      raise Element_Not_Found ;
      return This_Stack.Contents(This_Stack.Top) ;
   end if ;
end Top_Of ;
```

```
separate (Bounded Stack)
function Peek (At Element
                             : in Natural ;
                In_This_Stack : in Stack) return Element is
   -- returns the nth element below the top element in In_This_Stack
   -- (n is equal to At_Element). If At_Element is greater than the
   -- current number of elements in In_This_Stack, exception
   -- Element Not Found will be raised (note: Element Not Found will
   -- also be raised if In This Stack is empty)
   -- Written by Johan Margono, reviewed by (in alphabetical order):
   -- B.D. Balfour, E.V. Berard, G.E. Russell
   -- Author
                   Revision
                             Date
                                           Reason
                   1.0
                             8 Jul 1985
   -- J. Margono
                   2.0
                            19 Mar 1985
                                           Constraint_Error will not
   -- J. Margono
                                           be raised if pragma SUPPRESS
                                           is used
    - (c) 1986 EVB Software Engineering, Inc.
begin -- Peek
   if At_Element > In_This_Stack.Top then
      raise Element_Not_Found ;
      return In_This_Stack.Contents(In_This_Stack.Top - At_Element + 1) ;
   end if;
end Peek ;
```

```
〇
```

```
separate (Bounded_Stack)
procedure Clear (This_Stack : in out Stack) is
   -- set This_Stack to be an empty stack
   -- this is the same as if all elements had been popped off.
   -- Written by Johan Margono, reviewed by (in alphabetical order):
   -- B.D. Balfour, E.V. Berard, G.E. Russell
   -- Author
                   Revision
                                    8 Jul 1985
   -- J. Margono
                   1.0
   -- (c) 1986 EVB Software Engineering, Inc.
begin -- Clear
  -- re-assign stack top
  This_Stack.Top := Empty_Stack_Index ;
end Clear ;
```

```
separate (Bounded_Stack)
procedure Copy (This Stack : in
                                    Stack ;
                          :
                                  out Stack) is
   -- copy the contents of This_Stack into Into;
   -- will raise Overflow if the destination stack (Into) has
   -- length that is smaller than the number of elements in the
   -- source stack (This_Stack)
    -- Written by Johan Margono, reviewed by (in alphabetical order):
   -- B.D. Balfour, E.V. Berard, G.E. Russell
   -- Author
                    Revision
                                     Date
                                                        Reason
                                     8 Jul 1985
                    1.0
   -- J. Margono
   -- (c) 1986 EVB Software Engineering, Inc.
begin -- Copy
   if This_Stack.Top > Into.Maximum_Length then
      raise Overflow;
          -- there is enough room to put the contents of
          -- This_Stack into Into
      Into.Top := This_Stack.Top;
      Into.Contents(1 .. This_Stack.Top) :=
         This_Stack.Contents(1 .. This_Stack.Top);
   end if;
end Copy ;
```

3.4.3 Stepwise Refinement of the Other Program Units

None required.

3.4.4 Recursive Application of OOD

None required.

Test Programs

(T)

(

```
with Text IO ;
with Bounded Stack;
procedure Bounded_Stack_First_Test is
   -- This procedure is a driver to test the bounded stack package.
   -- It allows all operations to be invoked in any order.
   -- Written by J. Margono, and reviewed by B. D. Balfour, E. V. Berard,
   -- and G. E. Russell.
   -- Author
                     Revision
                                  Date
                                               Reason
   -- J. Margono
                      1.0
                                  8 Jul 1985
   -- J. Margono
                                 20 Feb 1986
                      2.0
                                               to make it uniform with
                                               all other first tests
    - (c) 1986 EVB Software Engineering, Inc.
   type Command is
     ( Clear,
                -- a stack
       Copy,
                 -- a stack to another stack
       Empty,
                -- is a stack empty?
      Equal,
                -- are two stacks equal?
      Elements, -- in a stack
      Peek,
                 -- at an element in a stack
                -- an element off a stack
      Pop,
                -- an element into a stack
      Push,
                -- top element of a stack
      Top,
                -- a stack
       View,
       -- command on the test program
       Quit
                -- quit the test
   package Command IO is new Text IO.Enumeration_IO(Enum => Command);
   package Integer_IO is new Text_IO.Integer_IO(Num => Integer) ;
   package Bounded is new Bounded Stack(Element => Integer) ;
      -- Integer is chosen simply to represent an enumeration data type
   function "=" (Left : in Bounded.Stack ;
                 Right: in Bounded.Stack) return Boolean
               renames Bounded. "=" ; -- makes "=" directly visible
   type Access To Stacks is access Bounded. Stack ;
   type Array_Of_Stacks is array (Positive range <>) of Access_To_Stacks ;
  Number Of Stacks : Positive ;
   procedure Display (This Stack : in Bounded.Stack) is separate ;
©1986 EVB Software Engineering, Inc.
```

```
    displays the contents of a given stack

begin -- Bounded_Stack_First_Test
   Get_Number_Of_Stacks:
   loop
      begin
         Text_IO.Put("How many STACK(s) do you need? ") ;
         Integer_IO.Get(Item => Number Of Stacks) ;
         exit Get_Number_Of_Stacks ; -- when there is no error
      exception
         when Text IO.Data Error =>
            Text IO.Skip Line ;
            Text_IO.Put_Line("Enter a POSITIVE number only!!!") ;
         when Constraint Error =>
            Text_IO.Put_Line("Enter a POSITIVE number only!!!") ;
         when others =>
            Text_IO.Put_Line("Unknown exception raised. Re-enter.") ;
      end;
   end loop Get_Number_Of_Stacks ;
   declare
     Stacks
                          : Array_Of_Stacks(1 .. Number_Of_Stacks) ;
      subtype Stack_Range is Natural range 1 .. Number_Of_Stacks ;
     Stack_Index
                         : Stack_Range := 1 ;
     Stack_Size
                         : Positive ;
     User_Command
                         : Command ;
     User Element
                         : Integer ;
     Stack_Number
                         : Positive ;
     Second Stack Number : Positive ; -- used in Copy and "="
     Element_Index
                         : Positive ; -- used in Peek
   begin
     Get_Stack_Sizes:
      loop
         begin
           Text IO.Put("Enter size for stack #") ;
           Integer_IO.Put(Item => Stack_Index, Width => 0) ;
           Text_IO.Put(" : ") ;
           Integer_IO.Get(Item => Stack_Size) ;
            -- declare the actual stack
           Stacks(Stack_Index) :=
```

```
new Bounded.Stack(Maximum_Length => Stack_Size) ;
            exit Get_Stack_Sizes when Stack_Index = Number_Of_Stacks ;
            Stack_Index := Stack_Index + 1 ;
         exception
            when Text_IO.Data_Error =>
               Text_IO.Skip_Line ;
               Text_IO.Put_Line("Enter a POSITIVE number only!!!") ;
            when Constraint_Error =>
               Text_IO.Put_Line("Enter a POSITIVE number only!!!") ;
            when others =>
               Text_IO.Put_Line("Unknown exception raised. Re-enter.") ;
         end :
      end loop Get Stack Sizes ;
     Test Stack :
      100p
         begin
           Text_IO.Put_Line("Selections :") ;
           Text_IO.Put_Line(" STACK");
           Text_IO.Put_Line("
                                   Clear,
                                             Copy, Empty, Equal; ") ;
           Text_IO.Put_Line("
                                   Elements, Peek, Pop,
           Text_IO.Put_Line("
                                   Top,
                                             View");
           Text_IO.Put_Line("
                                TEST PROGRAM") ;
           Text_IO.Put Line("
                                   Quit");
           Text_IO.Put("Enter selection : ") ;
           Command_IO.Get(Item => User_Command) ;
           exit Test_Stack when User Command = Quit ;
            case User_Command is
                when Push =>
                  Text_IO.Put("element : ") ;
                  Integer_IO.Get(Item => User Element) ;
                  Text_IO.Put("stack (1-") ;
                  Integer_IO.Put(Item => Number Of Stacks, Width => 0) ;
                  Text_IO.Put(") : ") ;
                  Integer_IO.Get(Item => Stack_Number) ;
                  Bounded.Push
                     (This Element
                                      => User_Element,
                       Into_This_Stack => Stacks(Stack_Number).all) ;
               when Pop =>
                  Text_IO.Put("stack (1-") ;
                  Integer_IO.Put(Item => Number_Of_Stacks, Width => 0) ;
                  Text_IO.Put(") : ") ;
                  Integer_IO.Get(Item => Stack Number) ;
©1986 EVB Software Engineering, Inc.
```

 $\xi_{\rm KF}$

283

```
Bounded.Pop (Top Element
                              => User Element,
               Off_This_Stack => Stacks(Stack_Number).all);
   Text_IO.Put("Top element was ") ;
   Integer_IO.Put(Item => User_Element, Width => 0) ;
   Text_IO.New_Line ;
when Empty =>
   Text_IO.Put("stack (1-") ;
   Integer_IO.Put(Item => Number_Of_Stacks, Width => 0) ;
   Text_IO.Put(") : ") ;
   Integer_IO.Get(Item => Stack_Number) ;
   if Bounded. Is Empty
          (This_Stack => Stacks(Stack_Number).all) then
      Text_IO.Put_Line("That stack is empty.") ;
      Text_IO.Put_Line("That stack is NOT empty.") ;
   end if;
when Elements =>
   Text_IO.Put("stack (1-") ;
   Integer_IO.Put(Item => Number_Of_Stacks, Width => 0) ;
   Text_IO.Put(") : ") ;
  Integer_IO.Get(Item => Stack_Number) ;
  Text_IO.Put("Number of elements in that stack is ") ;
  Integer_IO.Put
      (Item => Bounded.Number_Of_Elements
                (In_This_Stack => Stacks(Stack Number).all),
      Width => 0) ;
  Text_IO.New_Line ;
when Top =>
  Text_IO.Put("stack (1-") ;
  Integer_IO.Put(Item => Number_Of_Stacks, Width => 0) ;
  Text_IO.Put(") : ") ;
  Integer_IO.Get(Item => Stack_Number) ;
  Text_IO.Put("Top element is ") ;
  Integer_IO.Put
     (Item => Bounded.Top Of
                (This_Stack => Stacks(Stack_Number).all),
      Width => 0) ;
  Text_IO.New_Line ;
when Peek =>
  Text IO.Put("stack (1-");
  Integer_IO.Put(Item => Number Of Stacks, Width => 0) ;
  Text_IO.Put(") : ") ;
  Integer_IO.Get(Item => Stack Number) ;
  Text_IO.Put("Number of elements in that stack is ") ;
  Integer IO.Put
      (Item => Bounded.Number Of Elements
                (In_This_Stack => Stacks(Stack_Number).all),
      Width => 0) ;
  Text_IO.New_Line ;
```

```
Text_IO.Put("which element? ") ;
   Integer_IO.Get(Item => Element_Index) ;
   Text_IO.Put("Peeked element is ") ;
   Integer_IO.Put
      (Item => Bounded.Peek
                (At_Element
                               => Element Index,
                  In_This_Stack => Stacks(Stack Number).all),
       Width => 0) ;
   Text_IO.New_Line :
when Equal =>
   Text_IO.Put("first stack (1-") ;
   Integer_IO.Put(Item => Number_Of Stacks, Width => 0) ;
   Text_IO.Put(") : ") ;
   Integer_IO.Get(Item => Stack_Number) ;
   Text IO.Put("second stack (1-");
   Integer_IO.Put(Item => Number_Of_Stacks, Width => 0) ;
   Text IO.Put(") : ") ;
   Integer_IO.Get(Item => Second_Stack_Number) ;
   if Stacks(Stack_Number).all =
          Stacks (Second Stack Number) .all then
     Text_IO.Put_Line("Those stacks are equal.") ;
      Text_IO.Put_Line("Those stacks are NOT equal.") ;
   end if;
when Clear =>
   Text_IO.Put("stack (1-");
  Integer_IO.Put(Item => Number_Of_Stacks, Width => 0) ;
   Text_IO.Put(") : ") ;
   Integer_IO.Get(Item => Stack_Number) ;
   Bounded.Clear(This_Stack => Stacks(Stack_Number).all) ;
when Copy =>
  Text_IO.Put("source stack (1-") ;
  Integer_IO.Put(Item => Number_Of_Stacks, Width => 0) ;
  Text_IO.Put(") : ") ;
  Integer_IO.Get(Item => Stack_Number) ;
  Text IO.Put("destination stack (1-") ;
  Integer_IO.Put(Item => Number_Of_Stacks, Width => 0) ;
  Text_IO.Put(") : ") ;
  Integer_IO.Get(Item => Second_Stack_Number) ;
  Bounded.Copy
      (This_Stack => Stacks(Stack_Number).all,
       Into
                 => Stacks(Second_Stack_Number).all);
when View =>
  Text IO.Put("stack (1-");
  Integer_IO.Put(Item => Number_Of Stacks, Width => 0) ;
  Text_IO.Put(") : ") ;
  Integer_IO.Get(Item => Stack_Number) ;
```

t_w

```
Display(This_Stack => Stacks(Stack Number).all) ;
                 when others =>
                   Text_IO.Put("This command : ") ;
                   Command_IO.Put(Item => User_Command) ;
                   Text_IO.Put_Line(" is not implemented.") ;
             end case ;
          exception
            when Bounded.Overflow =>
               Text_IO.Put_Line("OVERFLOW was raised.") ;
            when Bounded. Element Not Found =>
               Text_IO.Put_Line("ELEMENT_NOT FOUND was raised.") ;
            when Bounded. Underflow =>
               Text_IO.Put_Line("UNDERFLOW was raised.") ;
            when Text_IO.Data_Error =>
               Text_IO.Put_Line("Incorrect command. Reenter.") ;
             when others =>
               Text_IO.Put_Line("Unknown exception raised. Reenter.") ;
         end ;
      end loop Test_Stack ;
   end;
exception
   when others =>
      Text_IO.Put_Line("Unknown exception reached the main program.") ;
      Text_IO.Put_Line("PROGRAM EXECUTION IS TERMINATED.") ;
end Bounded_Stack_First_Test ;
```

```
separate (Bounded_Stack_First_Test)
procedure Display (This_Stack : in Bounded.Stack) is
     displays the contents of a given stack
   -- Written by J. Margono, and reviewed by B. D. Balfour, E. V. Berard,
   -- and G. E. Russell.
   -- Author
                       Revision
                                   Date
                                              Reason
   -- J. Margono
                       1.0
                                   22 Jan 1986
   -- (c) 1986 EVB Software Engineering, Inc.
begin -- Display
   Text_IO.Put_Line("Stack contents :") ;
   Display Elements:
   for Index in 1 .. Bounded.
                     Number_Of_Elements
                      (In_This_Stack => This_Stack) loop
      Integer_IO.Put(Item => Bounded.
                              Peek (At_Element
                                                => Index,
                                  In_This_Stack => This_Stack),
                     Width => 0) ;
      Text_IO.New_Line ;
   end loop Display_Elements ;
end Display;
```

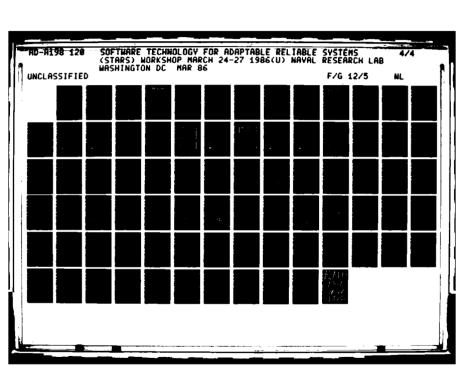
绘

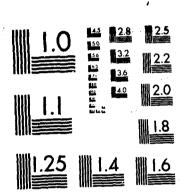
(Egr

```
with Bounded Stack;
with Text_IO;
procedure Second_Test_Of_Bounded_Stack is
   -- This program unit shows how a stack data structure can be used to
   -- evaluate simple POSTFIX expression (also known as REVERSE POLISH).
   -- Second_Test_Of Bounded Stack will first prompt the user for a file
   -- name of a data file that contains postfix expression separated by
   -- carriage returns. These expressions should only contain single-
   -- digit numbers and the following operators: "+", "-", "*", "/", and
   -- "$" (exponentiation). Also, there should not be any SPACES between
   -- operators or operands. Examples of valid postfix expressions :
            1. 98+42*/89-+
                              (i.e., (i.9+8)/(4*2))+(8-9))
            2. 344+*
                              (i.e., (4+4)*3)
   -- Second Test Of Bounded Stack will output the following after each
   -- successive iteration through the main loop (see code below):
   -- CURRENT SYMBOL read from the input, VALUE OF LEFT OPERAND,
   -- VALUE OF RIGHT OPERAND, and CONTENTS OF STACK.
   -- Written by Johan Margono and reviewed by B. Balfour, E. Berard,
   -- and G. Russell.
     Version
                       Author
                                         Date
                                                            Reason
   -- 1.0
                        J. Margono
                                         8 Jul 1985
    - (c) 1986 EVB Software Engineering, Inc.
                                                (Element => Integer) ;
  package New_Stack is new Bounded_Stack
  package Number_IO is new Text_IO.Integer IO (Num
                                                       => Integer) ;
  Operand_Stack : New_Stack.Stack (Maximum Length => 80) ;
  Left_Operand : Integer := 0 ;
  Right Operand : Integer := 0 ;
  Symbol .
             : Character ;
  Result
                : Integer := 0 ;
  File Name
               : String (1 .. 80) ;
  Length
               : Natural ;
  My_File
               : Text_IO.File_Type ;
  procedure Display_Contents_Of (This_Stack : in New_Stack.Stack) is
     -- displays the contents of a stack showing stack elements from
      -- top to bottom
  begin -- Display_Contents_Of
     for Element Index in 1 ...
              New Stack. Number Of Elements
                  (In_This_Stack => This_Stack) loop
        Number IO.Put(Item => New_Stack.Peek(At_Element
                                                           => Element Index,
                                              In_This Stack => This Stack),
                      Width => 3) ;
        Text IO.Put(" ") ;
      end loop ;
```

```
end Display_Contents_Of ;
    function Symbol_To_Natural (Symbol : in Character) return Natural is
      -- Returns the integer representation of a character which
      -- represents a decimal digit.
   begin -- Symbol_To_Natural
     return Character'Pos(Symbol) - Character'Pos('0') ;
   end Symbol_To_Natural ;
    function Is_Digit (Symbol : in Character) return Boolean is
      -- Returns True if the argument is a character which represents
      -- a decimal digit.
   begin -- Is Digit
      return Symbol in '0' .. '9';
   end Is_Digit ;
begin -- Second_Test_Of_Bounded_Stack
   Text_IO.Put
                    (Item => "Enter file name : ") ;
   Text_IO.Get_Line (Item => File_Name, Last => Length) ;
                    (File => My_File,
   Text_IO.Open
                     Mode => Text_IO.In_File,
                     Name => File_Name(1 .. Length)) ;
   while not Text_IO.End_Of_File (File => My File) loop
      -- display header
      Text_IO.New_Line ;
      Text_IO.Put
                       (Item => "SYMBOL") ;
      Text IO.Set Col (To => 15) ;
      Text IO.Put
                      (Item => "LEFT OPERAND") ;
      Text_IO.Set_Col (To
                            => ·30) ;
      Text IO.Put
                       (Item => "RIGHT OPERAND") ;
                            => 45) ;
      Text IO.Set Col (To
      Text IO.Put
                       (Item => "RESULT") ;
      Text_IO.Set_Col (To
                            => 60) ;
      Text_IO.Put_Line (Item => "STACK") ;
      while not Text_IO.End_Of_Line(File => My_File) loop
         Text_IO.Get (File => My_File, Item => Symbol) ;
         if Is_Digit(Symbol => Symbol) then
            New Stack.Push (This Element
                                          => Symbol To Natural(Symbol),
                           Into_This_Stack => Operand_Stack) ;
         else -- symbol must be an operator
            New Stack.Pop (Top Element => Right Operand,
©1986 EVB Software Engineering, inc.
```

```
Off_This_Stack => Operand_Stack) ;
                                   => Left_Operand,
      New_Stack.Pop (Top_Element
                      Off_This_Stack => Operand_Stack) ;
      case Symbol is
          when '+' =>
             Result := Left_Operand + Right_Operand ;
          when '-' =>
            Result := Left_Operand - Right_Operand ;
          when '*' =>
             Result := Left_Operand * Right_Operand ;
          when '/' =>
             if Right_Operand /= 0 then
               Result := Left_Operand / Right_Operand ;
             else
               Result := 0 ;
             end if ;
          when '$' =>
            Result := Left_Operand ** Right_Operand ;
          when others =>
             null;
     . end case ;
      New_Stack.Push (This_Element
                                      => Result,
                       Into_This_Stack => Operand Stack) ;
   end if;
   Text_IO.Put
                        (Item => Symbol) ;
   Text_IO.Set_Col
                        (To => 15);
   Number_IO.Put
                        (Item => Left_Operand, Width => 3) ;
   Text_IO.Set_Col
                        (To => 30);
   Number_IO.Put
                        (Item => Right Operand, Width => 3) ;
   Text_IO.Set_Col
                        (To => 45);
   Number_IO.Put
                        (Item => Result, Width => 3) ;
   Text IO.Set Col
                        (To => 60);
   Display_Contents_Of (This_Stack => Operand_Stack) ;
   Text IO.New Line ;
end loop ;
Text_IO.Skip_Line (File => My_File) ;
                   (Top Element => Result,
New Stack.Pop
                   Off_This_Stack => Operand_Stack) ;
(Item => "Final result : ") ;
Text IO.Put
Number IO.Put
                   (Item => Result) ;
Text_IO.New_Line ;
-- re-initialize operands and result
Left_Operand := 0 ;
Right_Operand := 0 ;
```





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A



```
Result := 0;
end loop;
Text_IO.Close (File => My_File);
end Second_Test_Of_Bounded_Stack;
```



Bibliography

[EVB, 1985]. EVB Software Engineering, An Object Oriented Design Handbook for Ada Software, EVB Software Engineering, Rockville, Maryland, 1985.

[Knuth, 1973]. D.E. Knuth, The Art of Computer Programming, Volume 1: Fundamental Algorithms, Second Edition, Addison-Wesley, Reading, Massachusetts, 1973.

Bibliography

- [Alexandridis, 1986]. N.A. Alexandridis, "Adaptable Software and Hardware: Problems and Solutions," Computer, Vol. 19, No. 2, February 1986, pp. 29 39.
- [Bauer and Wossner, 1982]. F.L. Bauer and H. Wossner, Algorithmic Language and Program Development, Springer-Verlag, New York, New York, 1982.
- Blank and Krijger, 1983]. J. Blank and M.J. Krijger, Editors, Software Engineering: Methods and Techniques, John Wiley & Sons, New York, New York, 1983.
- [Boar, 1984]. B.H. Boar, Applications Prototyping, John Wiley & Sons, New York, New York, 1984.
- [Boehm-Davis and Ross, 1984]. D. Boehm-Davis and L.S. Ross, "Approaches to Structuring the Software Development Process," General Electric Company Report Number GEC/DIS/TR-84-B1V-1, October 1984.
- [Booch, 1982]. G. Booch, "Object Oriented Design," Ada Letters, Vol. I, No. 3, March-April 1982, pp. 64 76.
- [Booch, 1983] G. Booch, Software Engineering with Ada, Benjamin/Cummings, Menlo Park, California, 1983
- [Booch, 1985]. G. Booch, "Dear Ada," Ada Letters, Vol. IV, No. 6, May-June 1985, pp. 21 26.
- [Campos and Estrin, 1978]. I.M. Campos and G. Estrin, "SARA Aided Design of Software for Concurrent Systems," in AFIPS Conference Proceedings, Vol. 47, 1978.
- [Dijkstra, 1968]. E.W. Dijkstra, "Structure of the THE'-Multiprogramming System," Communications of the ACM, Vol. 11, No. 5, May 1968, pp. 341-346.
- [DOD, 1978]. Department of Defense Requirements for High Order Computer Programming Languages: "Steelman", NTIS Order Number ADA059444, 1978.
- [DOI, 1981]. British Department of Indudtry, Report of the Study of an Ada-Based System Development Methodology, Department of Indudtry (UK), 1981.
- [EVB, 1986] EVB Software Engineering, An Object Oriented Design Handbook for Ada Software, 1985
- [Freeman and Wasserman, 1982]. P. Freeman and A. I. Wasserman, Software Development Methodologies and Ada, Department of Defense Ada Joint Program Office, 1982.
- [Goguen, 1986] J.A. Goguen, "Reusing and Interconnecting Software Components," Computer, Vol. 19, No. 2, February 1986, pp. 16 28.
- ©1986 EVB Software Engineering, Inc.

- [Hanson, 1983]. K. Hanson, Data Structure Program Design, Ken Orr and Associates, Topeka, Kansas, 1983.
- [Hibbard et al, 1983]. P. Hibbard, A. Hisgen, J. Rosenberg, M. Shaw, and M. Sherman, Studies in Ada Style, 2nd. Edition, Springer-Verlag, New York, New York, 1983.
- [IEEE, 1983]. IEEE, IEEE Standard Glossary of Software Engineering Terminology, The Institute of Electrical and Electronic Engineers, New York, 1983.
- [Jackson, 1983]. M. Jackson, System Development, Prentice-Hall, Englewood Cliffs, New Jersey, 1983.
- [Jackson, 1985]. M.I. Jackson, "Developing Ada Programs Using the Vienna Development Method (VDM)," Software Practice and Experience, Vol. 15, No. 3, March 1985, pp. 305 318.
- [Jones, 1980]. C.B. Jones, Software Development: A Rigorous Approach, Prentice-Hall, Englewood Cliffs, New Jersey, 1980.
- [Jones, 1983]. T.C. Jones, Editor, Tutorial: Programmer Productivity: Issues for The Eighties, IEEE Catalog Number EHO186-7, Computer Society Order Number 391.
- [Kernighan and Plauger, 1978]. B.W. Kernighan and P.J. Plauger, The Elements of Programming Style, 2nd Ed., McGraw-Hill Book Company, New York, New York, 1978.
- [Knuth, 1973]. D.E. Knuth, The Art of Computer Programming, Volume 1/Fundamental Algorithms, 2nd Ed., Addison-Wesley, Reading, Massachusetts, 1973.
- [Knuth, 1974] D.E. Knuth, "Structured Programming with GOTO's", Current Trends in Programming Methodology Vol. 1, Prentice-Hall, Englewood Cliffs, New Jersey, 1977
- [Ledgard, 1975] H.F. Ledgard, *Programming Proverbs*, Hayden Book Company, Rochelle Park, New Jersey, 1975
- [Masters and Kuchinski, 1983]. M.W. Masters and M.J. Kuchinski, "Software Design Prototyping Using Ada," Ada Letters, Vol. II, No. 4, January-February 1983, pp. 68 75.
- [Myers, 1976]. G.J. Myers, Software Reliability: Principles and Practices, John Wiley & Sons, New York, New York, 1976.
- [Myers, 1978]. G.J. Myers, Composite/Structured Design, Van Nostrand Reinhold, New York, New York, 1978.
- [Newsted et. al, 1981] P. Newsted, W. K. Long, J. Yeung, "The Impact of Programming Styles on Debugging Efficiency", ACM SIGSOFT Software Engineering Notes, Vol. 6, No. 5, 1981
- [Nissen and Wallis, 1984]. J. Nissen and P. Wallis, *Portability and Style In Ada*, Cambridge University Press, Cambridge, United Kingdom, 1984.
- ©1986 EVB Software Engineering, Inc.

- [Parnas, 1972]. D.L. Parnas, "On the Criteria To Be Used in Decomposing Systems Into Modules," Communications of the ACM, Vol. 5, No. 12, December 1972, pp. 1053-1058.
- [Ross et al, 1975]. D. T. Ross, J. B. Goodenough, and C. A. Irvine, "Software Engineering: Process, Principles, and Goals," Computer, May 1975, pp. 65 75.
- [St. Dennis et al, 1986] R. St. Dennis, P. Stachour, E. Frankowski, and E. Onuegne, "Measurable Characteristics of Reusable Ada Software," Ada Letters, Vol. VI, No. 2, March, April 1986, pp. 41 50.
- [Shankar, 1982]. K.S. Shankar, "A Funtional Approach to Module Verification," *IEEE Transactions on Software Engineering*, Vol. SE-8, No. 2, March 1982, pp. 147-160.
- [Wirth, 1971]. N. Wirth, "Program Development by Stepwise Refinement," Communications of the ACM, April, 1971 pp. 221 227. Reprinted in E. Yourdon (ed), Writings of the Revolution, Yourdon Press, New York, 1982.
- [Yourdon and Constantine, 1979]. E. Yourdon and L.L. Constantine, Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design, Prentice-Hall, Englewood Cliffs, New Jersey, 1979.



Government Information Systems Division Software Operation

STARS Applications Systems and Reusability Workshop

Harris Corporation

March 24-27, 1986

Automated Measurement System (AMS)

Environment Measurement Instrumentation (EMI)

Presenter: Karen L. Like

M HARRIS

Government Information Systems Division Software Operation

AUTOMATION OF THE SOFTWARE QUALITY FRAMEWORK

- AUTOMATION IS NEEDED TO: 0
- O MAKE THE APPLICATIONS OF THE FRAMEWORK TO LARGE PROJECTS REASONABLE IN TERMS OF:
- COST TO COLLECT DATA

- COST OF ANALYSIS OF DATA ABILITY TO VERIFY DATA (BY IV&V OR CONTRACTING AGENCY) TIMELINESS OF RESULTING REPORTS
 - WILLINGNESS OF CONTRACTORS TO PROPERLY APPLY THE FRAMEWORK
- O ENSURE CONSISTENCY OF COLLECTED DATA

HARRIS

Government Information Systems Division Software Operation

THE AMS TOOL EFFORT

AN AUTOMATED DATA COLLECTION AND ANALYSIS TOOL WHICH AUTOMATES THE RADC QUALITY

FRAMEWORK AND PROVIDES THE FOLLOWING:

- FULL LIFE CYCLE SUPPORT
- REQUIREMENTS PHASE SREM
 - DESIGN PHASE SDDL
 - CODING PHASE SAP
- FLEXIBILITY USER CONTROL OF FRAMEWORK
- HUMAN ENGINCERED MENU-DRIVEN SYSTEMS
 INTERCONNECTION WITH OFF-THE-SHELF DESIGN AND REQUIREMENTS TOOL
- AUTOMATED DATA COLLECTION OF FORTRAN AND ADA SOURCE CODE
- MANAGEMENT AND ANALYSIS REPORTS REGARDING THE DEVELOPING SOFTWARE
- PORTABILITY-TARGETED FOR VAX 11/780 AND IBM PC/AT

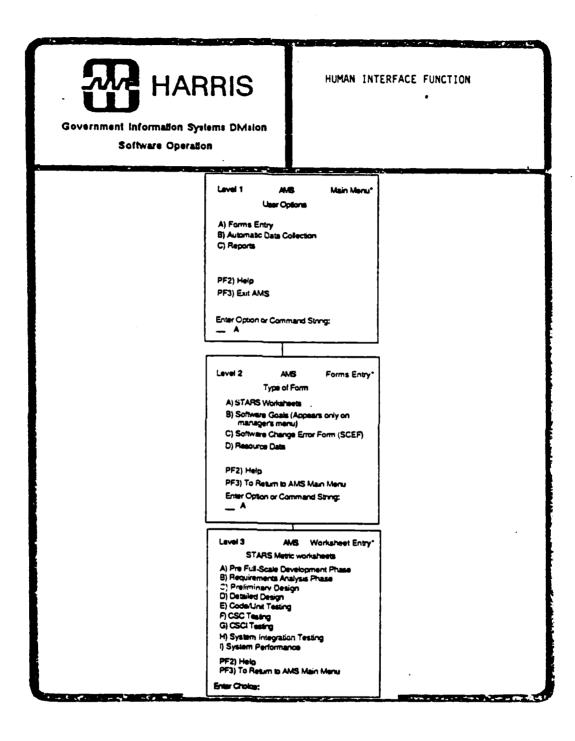


Government information Systems DMsion Software Operation

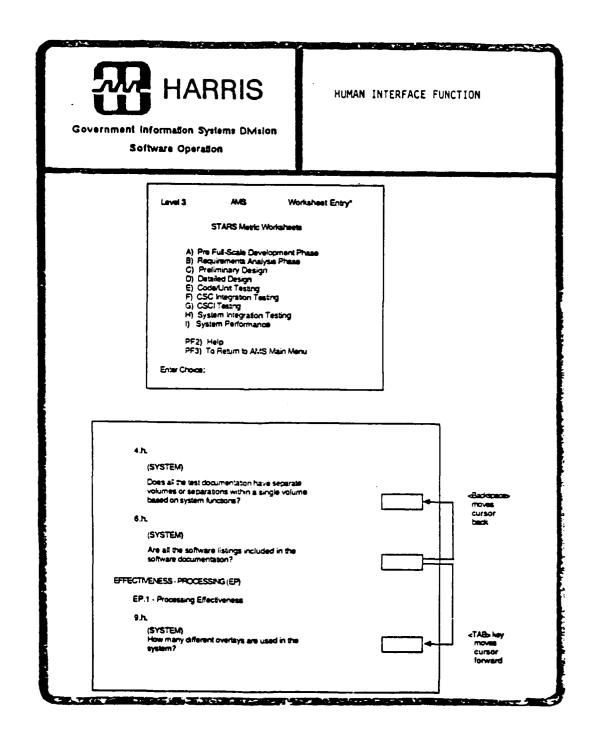
AMS DESIGN EMPHASIZES

- o Flexibility and Adaptability
- o Automation
- o Reusability
 - Human Interface Function
 - Screen based Application Support System
 - Data base management function
 - RIM5
 - Automatic Data Collection Function
 - Source Analyzer Program (SAP)
 - Report Generator Function
 - U-CAN USE Graphics Package













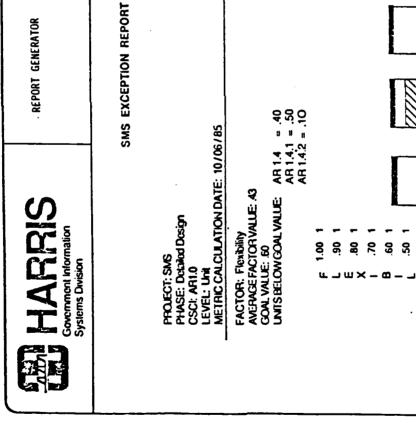
REPORT GENERATOR

SMS GOAL REPORT

PROJECT: SMS
PHASE: Requirements
LEVEL: CSCI
ENTITY NAME: ARLO
METRIC CALCULATION DATE: 10/00

DATE: 12/05/85

ACTUAL	36.	.50	89.	55.	3 .	0	.45	27.	27.	27.	07.	59 .	09:
1V05	86.	26:	.93	.92	8 6:	89	8.	ЗГ.	37.	375	5 .	τ.	02.
FACTOR	Flexibility	Reusability	Portability	Maintainability	Usabiliny	Correctness	Reliability	Survivability	Verifiability	Expandability	Interoperability	Efficiency	Integrity

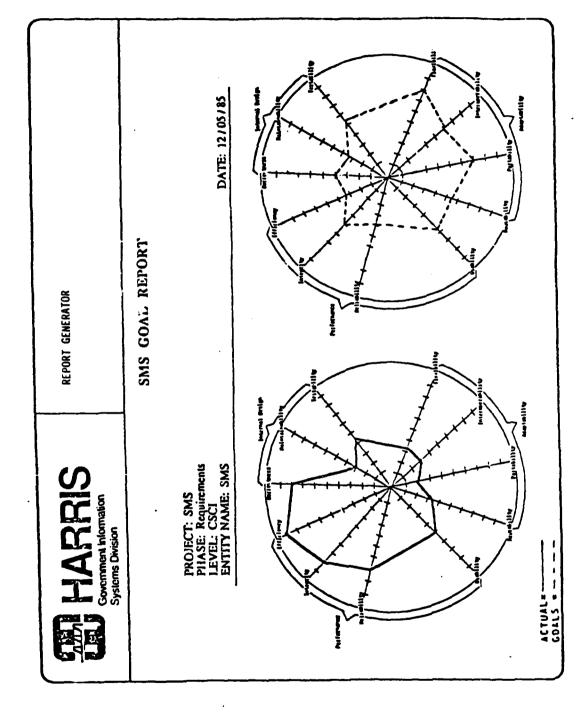


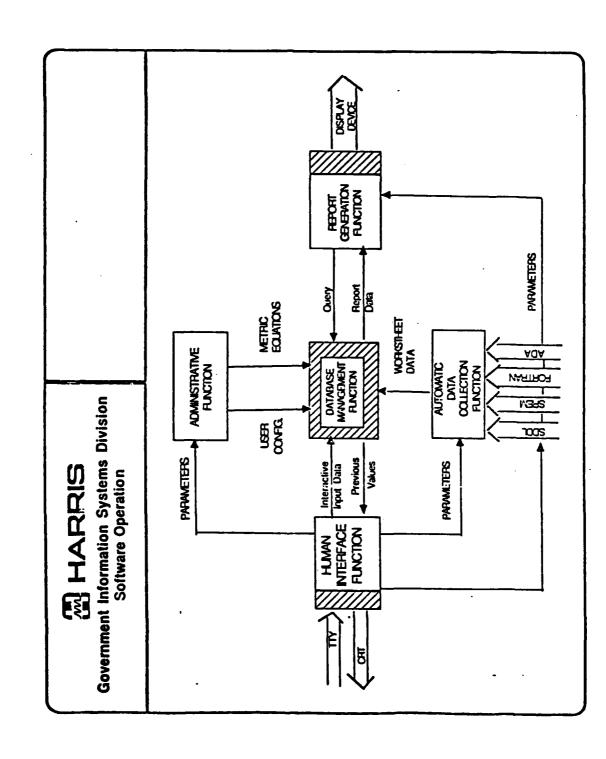
DATE: 12/05/85

CZZZZZZ = Actual

DATE: 12/05/85 12/1 00 SMS GROWTH REPORT REPORT GENERATOR 702 0 PROJECT: SMS
PHASE: Requirements - Detailed Design
LEVEL: CSCI
ENTITY NAME: AR1.2
MERIC CALCULATION DATE: 10/05/85 S/A REQ 0 HARPRIS Government Information Systems Division 315 .25 0.0 0.







B

∰)

HARRIS HARRIS

Government Information Systems Division Software Operation

PHASED RELEASES

- O PHASE 1 HUMAN INTERFACE AND DATABASE
- PHASE 2 REPORT GENERATOR
- PHASE 3 AUTOMATIC DATA COLLECTION AND CALCULATE METRICS
- PHASE 4 AOMINISTRATIVE

RELEASE DATES

SEPTEMBER 1985	NOVEMBER 1985	FEBRUARY 1986	APRIL 1986
PHASE 1	PHASE 2	PHASE 3	PHASE 4



今

C31 ENVIRONMENT MEASUREMENT INSTRUMENTATION

OBJECTIVES

DATA COLLECTION IN ACCORDANCE WITH STARS SOFTWARE EVALUATION REPORT. SPECIFY AND DESIGN NECESSARY INSTRUMENTATION MECHANISMS REQUIRED WITHIN A LIFE-CYCLE SOFTWARE ENGINEERING ENVIRONMENT TO SUPPORT BASIC:

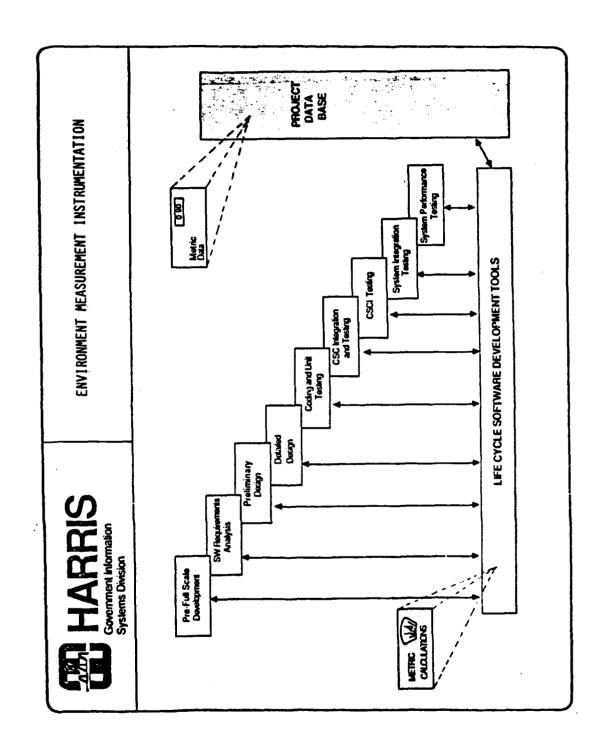
OPTION: SPECIFY DESIGN FOR:

SOFTWARE RESOURCE EXPENDITURE REPORT SOFTWARE TEST INFORMATION REPORT

SOFTWARE CHARACTERISTICS REPORT

SOFTWARE CHANGE/PROBLEM REPORT

SOFTWARE ENVIRONMENT REPORT





G

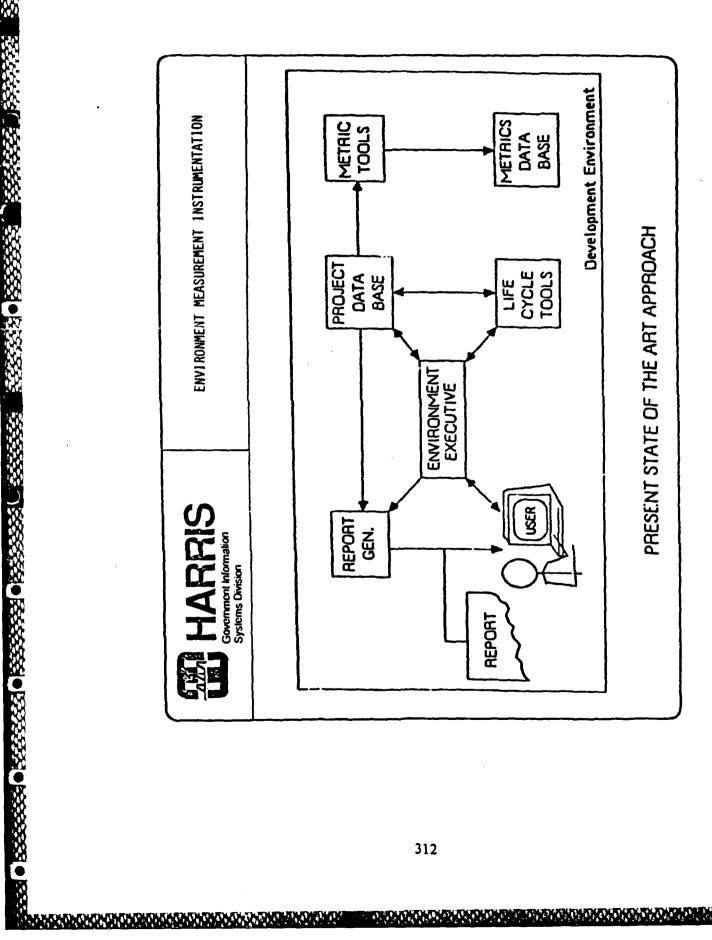


Government Information Systems Division
Software Operation

C3I ENVIRONMENT MEASUREMENT INSTRUMENTATION

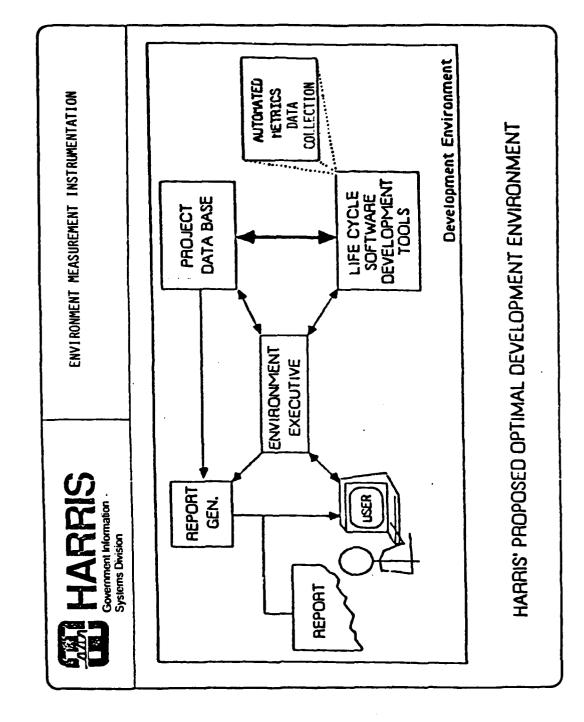
TASKS

- TASK I: EXAMINE SOFTWARE DEVELOPMENT LIFE-CYCLE FOR ORIGIN OF DATA ITEMS.
 - 1. AUTOMATABILITY ANALYSIS
 - 2. SCENARIO PREPARATION
- TASK II: DETERMINE AND DESCRIBE INSTRUMENTATION MECHANISMS
 ACROSS LIFE CYCLE.
 - 1. MECHANISM DETERMINATION
 - 2. DATABASE SIZING/ARCHITECTURE
 - 3. DATA ANALYSIS
- TASK III: TOP LEVEL DESIGN OF ALL DATA WRITTEN MECHANISMS
 - 1. OPERATIONAL CONCEPTS DOCUMENT
 - 2. SYSTEM REQUIREMENT SPECIFICATION
 - 3. SOFTWARE OPT LEVEL DESIGN DOCUMENT





EN





SUMMARY STATUS	58/6/8 Þ	Kickoff 9/17/85	Analysis Complete	t e	leport 27 November 85	Report 21 January 86	
Government Information Systems Division Software Operation	Program Initiated	Formal Customer Kickoff 9/17/85	Automatability A	Scenarios Complete	Task I Interim Report	Task I Final Rep	
Government Inform	•	•	•	•	•	•	

STARS BUSINESS PRACTICES MANAGEMENT WORKSHOP

DOD-STD-2167 SOFTWARE DEVELOPMENT STANDARDS
PACKAGE DEVELOPMENT AND OPEN ISSUES

Los Angeles, CA 18-22 November 1985

Ole Colubjatnikov
Chairman, EIA DOD-STD-SDS Review Committee
Vice-Chairman, CODSIA Task Group 21-83 on DOD-STD-SDS
General Electric Company
Syracuse, New York 13221
Phone: (315) 456-4744

CONTENTS

•	L	_	٠	_	_	_	4
^	D	4	Ŧ	r	2	r	٩

Biography

Introduction

Section I. Overview of the SDS Package and Development Process

- A. The Problem and SDS Objectives
- B. The SDS Package
- C. The SDS Package Development Stages
- D. A Broadly Based Public Review and Participation Process
- E. SDS Package Evolution
- F. The Issue Driven Approach

Section 2. The Issue Driven Approach (IDA)

- A. Defense Standards Requirements
- B. A Systematic Approach
- C. Due Process and Consensus Development
- D. Efficiency and Effectiveness of the Voluntary Standards Process
- E. Active Versus Reactive Standards Development
- F. Technology Transistioning From R&D to Battle Operational Environment

Section 3. SDS Development Issues

- A. Issue Status at the Beginning of CODSIA Review (Cycle 3)
- B. Issue Status at the Point of SDS Package Initial Release
- C. Limited Coordination and Proposed Changes

Section 4. SDS Implementation and Plans for Revision A

- A. Government DOD-STD-2167 Implementation Plans
- B. Industry DOD-STD-2167 Implementation Plans
- C. Assessment of SDS Interim Release
- D. SDS Revision A Overview and Milestones
- E. Summary of Open issues
- F. New Revision A Initiated Issues

Section 5. In Summary

- A. SDS Package Assessment
- **B.** Acknowledgements
- C. Conclusions
- D. Recommendations

References

DOD-STD-2167 SCFTWARE DEVELOPMENT STANDARDS PACKAGE DEVELOPMENT AND OPEN ISSUES

Ole Golubjatnikov
Chairman, EIA DOD-STD-SDS Review Committee
Vice-Chairman, CODSIA Task Group 21-83 on DOD-STD-SDS
General Electric Company
Syracuse, New York 13221
Phone: (315) 456-4744

ABSTRACT

This paper is based on extractions from the CODSIA Task Group 21-83 Report on the DOD-STD-2167(SDS) Package Coordination Review.

The development of DOD-STD-2167 Software Development Standards (SDS) Package is one of the most complex and comprehensive standards development efforts undertaken by the Department of Defense and the defense industry. The SDS Package development spans a 6 year period from April 1979 through June 1985 with implementation and revision efforts projected into the next decade. Over 300 individuals and 150 corporations and Government components participated in this joint effort consisting of three Government sponsored workshops, five industry (EIA) sponsored workshops, and three review cycles containing approximately 12,000 review comments. The DOD-STD-2167 development is a significant departure from a conventional defense standards development approach and can be used as a future model for improving standards development process in the mission critical computer resources area.

The evolution of the SDS Package is based on an Issue Driven Approach (IDA) which Jocuses on the root causes of the review comments rather than fixing on the apparent problem. IDA considers conflicting goals, as well as alternative methods of solution starting from raw comments, to root concerns, to basic issues. Once an issue is identified, it remains on the list, permitting traceability and follow-up to assure its continued resolution. This issue-driven approach makes the integration of conflicting factors manageable through an incrementally evolving negotiation process between DoD and industry representatives.

The DOD-STD-2167 Standards Package was released for DoD-wide use on 4 June 1985 and is based on final or interim solutions to 35 issues extracted from 12,000 raw comments. Out of this total, Revision A work is continuing on 18 issues with interim solutions, which require extensive research and development. Many of these open issues are identical to those being addressed by the other DoD software initiatives: Ada, the STARS Program, and the Software Engineering Institute (SEI). Therefore, it is recommended that these initiatives should participate in the Joint Logistic Commanders/Computer Resources Management (JLC/CRM) SDS initiative so as to accelerate the resolution of these longer term SDS issues.

This paper discusses the JLC SDS initiative and it elaborates on the issuedriven approach by describing the concept and providing the status and description of the 35 issues encountered during the DOD-STD-2167 development. This discussion also provides plans for the resolution of the Revision A open issues and the implementation of DOD-STD-2167 in the DoD and industry.

Copyright (c) 1985 by Ole Golubjatnikov and CODSIA Task Group 21-83

BIOGRAPHY

Ole Golubjatnikov is the chairman of the EIA DOD-STD-SDS Review Committee and the vice-chairman of the Council of Defense and Space Industry Associations (CODSIA) Task Group 21-83 on DOD-STD-SDS Software Development Standards. He was responsible for conceiving and introducing the issue-driven approach to defense standards. He is also the editor and principal contributor of the CODSIA Report on DOD-STD-2167 and has made numerous contributions to the standard, including Appendix D on tailoring and proposed Appendix E on system engineering integration of prime and critical items. Over the years, Ole has participated in practically all DoD mission critical computer and software initiatives in an industry leadership role. He is an EIA representative on ANSI X3 and a U.S. delegate to ISO/TC97/SC7 on design and documentation of Computer-Based Systems.

Mr. Galubjatnikov was the principal contributor and editor of the CODSIA Task Group 13-82 Report on DoD Management of Mission-Critical Computer Resources to USD (R&E) and the principal industry reviewer of the DOD Computer Technology Report to Congress and its Study Annex. He has contributed many innovative solutions to complex technical, management, and acquisition issues and has supported in setting the stage for a new direction in DoD computer policy to maintain United States' defense computer technology leadership. The above described CODSIA Task Group 13-22 recommendations and activities led to the formation of DoD's Computer Resources Council (CRC) and Defense Computer Resources Board (DCRB). Ole was also instrumental in the formation of the STARS Joint Industry Interface Working Group and the Computer Systems Interface Working Groups.

During recent years, he has participated as a member of CODSIA, EIA, NSIA, and Navy task groups, and is a senior member of IEEE, the Computer Society and ACM. As a result of his GE and industry association activities, he has reviewed plans or participated in practically all recent DoD computer technology, policy and standards initiatives.

Mr. Golubjatnikov is a consultant for data systems architecture and the management of computer resources. His current assignment includes the development and strategic planning of surface ship ASW systems and computer resources, and the development of GE's software support environment based on Ada and DOD-STD-2167. He has published numerous reports and computer and software conference papers.

During his career, he has been involved with more than one hundred real-time computer systems and distributed computer networks for commercial and military applications. He was responsible for the conception of the radar peripheral architecture of the AN/TPS-59 which is the baseline for GE's Solid State Radar Family. Prior to rejoining GE in 1977, he was president of COMPTA, Inc. and an independent consultant for a period of six years specializing in computer architecture and real-time data systems and software for industry, business, and defense applications.

Mr. Golubjatnikov has been associated with the computer field since 1950 as an undergraduate student at the University of Illinois, working on the ILLIAC and ORDVAC computers. He was the Manager of GE's M-600 military computer product line peripheral equipment engineering. He has been engaged in the development and design of twelve commercial and aerospace computers and associated peripherals and software products for GE and Honeywell, including M-236, M-605, M-625, MULTICS, 2416, MQX, DN-355, FCAN, MK-500, AOP, FFP and MCF.

INTRODUCTION

This paper will review the SDS process and the lessons learned as they relate to future defense standards developments. The issue-driven approach will be described, as well as a review and description of the plans for DOD-STD-2167 implementation and the resolution of Revision A open issues. Towards this goal, this paper is presented in the following five sections:

- 1. Section 1 provides an overview of the SDS Package development process.
- 2. Section 2 describes the issue-driven approach.
- Section 3 reviews the status of the 55 SDS development issues.
- Section 4 describes the Joint Logistics Commanders/Computer Software Management (JLC/CSM) Subgroup and Council of Defense and Space Industry Associations (CODSIA) Task Group 21-83 plans for the resolution of Revision A open issues and DOD-STD-2167 implementation.
- 5. Section 5 summarizes the results of the JLC SDS initiative, acknowledgements, conclusions, and recommendations.

SECTION I. OVERVIEW OF THE SDS PACKAGE AND ITS DEVELOPMENT PROCESS.

The SDS Package has evolved through 3 public review cycles, and 13 document versions with 12,000 comments and 55 issues addressed. It has been a massive effort based on the best of Government and industry voluntary standards participation and contractor's efforts. The total effort is estimated at \$10 million with a 50/50 split between voluntary efforts and Government funding. The process contains lessons learned and sets a standard for the improvement of future defense standards development efforts.

This section defines the defense software standards problem, provides an overview of the SDS development process and characterizes the evolving SDS product.

A. THE PROBLEM AND SDS OBJECTIVES

Mission-Critical Computer Resources (MCCR) are a key element within modern defense systems. Efficient and effective development and management of these resources is fundamental to system development, interoperability and longevity – three key factors which will determine the success of our defense in coming years and its cost.²

In the mid-1970s, studies were conducted by the DoD indicating serious performance problems, schedule slippages, and cost problems with practically all major weapon systems. Much of this was directly related to software associated with the defense systems.

Following these studies, a uniform computer resource management policy, DODD 5000.29 and DODI 5000.31, was introduced by OSD covering all DoD embedded computer applications. The implementation of this policy in MCCR management and the HOL development and usage enforcement has been reasonably successful. The other areas targeted for correction were: (1) improved coordination of DoD software R&D, which has since evolved into the STARS Program and the Software Engineering Institute, and (2) software development standards.

Current generation of DcD software development standards such as MIL-STD-1679 and MIL-STD-190 have evolved ad hoc over a period of two decades. A number of defense system and software problems in the 1970s and early 1980s are traceable to problems with software acquisition, development, and support policies and standards. These problems include:

- o Service and agency unique
- Inconsistent terminology and requirements
- o Neglect of various aspects of software acquisition development and support
- Incompatibility with modern methods of developing software
- o Prescribed requirements which are unsupported by documentation system
- o Requirements often established by implication

- d Requirements which are subjective and cannot be easily measured
- o Not designed for tailoring as a function of project size or software category

Conflicting, redundant, and in some cases, nonexistent software development, acquisition, and support policies and standards frequently result in:

- o Confusion in the program office
- Duplication of effort
- o Contractors maintaining multiple management systems
- o Adding unnecessary costs to the software acquisition process
- o Inability to focus and apply software R&D efforts and accelerate technology transfer and insertion

The JLC software standardization program objectives were jointly developed by DoD and industry participants during the Monterey I workshop. These objectives produced a complete and consistent set of tri-service software acquisition, development, and support policies and standards which:

- Establish a well-defined and easily understood software acquisition and development process
- o Provide adequate visibility during software development and acquisition
- o Reduce confusion and eliminate conflicts in existing standards
- o Are compatible with modern methods of developing software
- o Provide cost benefits over the entire life cycle
- o Increase probability of obtaining quality software

B. THE SDS PACKAGE

A multi-service group in the DoD, the Joint Logistics Commanders (JLC), is developing a new software development standards package. This package is the result of the initiative undertaken by JLC/Joint Policy Group on Computer Resource Management (JLC/CRM) in April 1979. The following actions were identified by the JLC Workshop, identified as Monterey I:

- 1. Develop a general tri-service policy framework for software acquisition that addresses the entire software life cycle and provides uniform terminology and definitions.
- 2. Develop uniform military standards for use by all services and agencies consistent with the policy framework.
- 3. Define and develop a comprehensive set of DIDs for all services and agencies which support the acquisition policy and standards.

The work initiated at Monterey I culminated in the release of the Software Development Standards (SDS) Package 4 June 1985 which consists of:

1. Joint Regulation, Management of Computer Resources in Defense Systems

- 2. DOD-STD-2167, Defense System Software Development
- 3. An integrated and tailorable set of 24 Data Item Descriptions (DIDs) grouped into four areas:
 - 5 management DIDs
 - 9 design documentation DIDs
 - 4 test documentation DIDs
 - 6 support documentation DIDs
- Updates to software aspects of the following three existing standards:
 - MIL-STD-483A, Configuration Management Practices for Systems, Equipment, Munitions and Computer Programs
 - o MIL-STD-490A, Specification Practices
 - MIL-STD-1521B, Technical Reviews and Audits for Systems, Equipments and Computer Programs

An additional component of the JLC software standards package is the draft DOD-STD-2168, Software Quality Evaluation and its 2 associated DIDs.

C. THE SDS PACKAGE DEVELOPMENT STAGES

During the evolution, from 1979 through 1985, the SDS Package progressed through the following stages and steps:

- 1. JLC Monterey I Workshop: April 1979
- 2. JLC Monterey II Workshop: June 1981
- 3. Draft DIDs (TRW) and standards (DRC) development: 1920-1922
- 4. Draft Review (Cycle 1): June 1982 to May 1983
 - First Government/industry review
 - EIA Dallas Workshop: September 1982
 - EIA and JLC/CSM review meetings
 - Document set rewrites
- 5. Select Panel Review (Cycle 2): May 1983 to January 1984
 - EIA and Select Panel Review: May 1983
 - Select Panel Meeting: May 1983
 - EIA Los Angeles Workshops June 1983
 - EIA Phoenix Workshop: September 1983
 - CODSIA Task Group 21-83 formation: September 1983
 - JLC Orlando I Workshop: October 1983
 - O Document set rewrites: June to December 1983
- 6. CODSIA Review (Cycle 3): January 1984 to June 1985
 - o Formal coordination review: January-April 1984

- CODSIA and JLC/CSM review meetings and workshops
- o EIA Tampa Workshop: September 1984
- o Document set rewrites: June 1984 to May 1985
- DMSSO review, approval and distribution: January to June 1985

The standards package dated 4 June 1985 was released for DoD-wide use in July. With the completion of Review Cycle 3 and release of the standards package, the following two parallel stages of DOD-STD-2167 development are underway:

- 7. DOD-STD-2167 Implementation in DoD and industry
 - a EIA St. Louis Workshop: September 1985
- \$ DOD-STD-2167 Revision A planning and development
 - o EIA St. Louis Workshop: September 1985

D. A BROADLY BASED PUBLIC REVIEW AND PARTICIPATION PROCESS

DOD-STD-2167 is applicable to the complete software life cycle and the full range of Jefense software applications. For example, it addresses defense software in full-scale development, as well as firmware and reusable or commercial software. It has many complex interfaces to related engineering disciplines, including project management, system engineering, configuration management, and quality evaluation. This complex scope and nature of DOD-STD-2167 dictates the need for broad public review and participation by all segments of DoD and industry affected by the standard.

The industry participation from Monterey I through CODSIA Review Cycle 3 is summarized in Table 1-1. While a total of 68 corporations participated in some of the 5DS development and review activities, six corporations (GE, IBM, Logicon, Sperry, TI, and TRW) participated in all six workshops and reviews. Boeing, DRC, Hughes, Singer, and SDC participated in five of the SDS activities. The corporations listed represent practically all major segments of defense software applications and a number participated through more than one industry association as indicated in Table 1-1. In some cases, different divisions of the same corporation are represented in different industry associations, and thus have submitted a unique set of review comments. In other cases, the same set of review comments was submitted to more than one industry association. These redundant comments were deleted during the review process and do not appear in the statistics. The CODSIA coordinated review process considered all comments regardless of source or statistical significance.

E. SOS PACKACE EYCLUTION

Thirteen drafts of the SDS Package were developed by the contractor (DRC). These drafts were reviewed by industry, DoD components, JLC/CSM Subgroup, CODSIA, DMSSO, and EIA SDS review committee during the three review cycles as summarized in Table 1-2.

During review cycle 1, the rework of the SDS Package by DRC was based on DoD components and industry detailed comments. Little use, if any, was made of

SUMMARY OF COAPORATE PARTICIPATION IN THE DOD-STD-2167 BDS) DEVELOPMENT AND REVIEW PROCESS TABLE 1-1

	Industry Association Review Comments	2 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5			SDS Review/	3DS Review/Development Stage	2.05		
Corporation	VW	EIA	MSIA	(6261) degrapa f falengy	Monterey 2 Workshop (1981)	Orlands Workshop	Cycle 1 Draft Revise (*ElA Warkshop)	Cycle 2 Select Panel Review	Cycle) Coordination Review (*CODSIA)
AAI Ad-arcad Technology			×		·	×			×
Aerojet Aerojede	×	×		,	×		X.	×	×
Archer Selvaine Bends Beshg Besh Aliza	×	××	. * *		××	×	\$	×	×××
Burrayas CACI CDC CSC		××		×	×	XXXX	£	*	×
DAC E-Systema Ford Aero General Dynamics		M	×××		×	×	× ××	××	××××
Cororal Electric Crumman CTE Higher Order Software	××	××	× ×	×	H K	××	: . : * * *	Å××	* * *
Honorell Hughes Dul	×××	×××	××		××	XXX	£ж	××	×××
HTRJ Princes Systems Pust ITT			X	×	XX X			×	· ×
Leav-Siagler Littee Lockheed Lesting		×××		××	××	×	* ×	*× ×	* ××



SUMMARY OF CORPORATE PARTICIPATION IN THE DOD-STD-2167 (SDS) DEVELOPMENT AND REVIEW PROCESS TABLE 1-1 (CONT.)

	A SK	Medity Inches							
	و ال	Revise Comments			SDS Review/	SDS Review/Development Stage	Stage		
Corporation				Monterey 1	Monterey ?	Orlanda	Cycle 1	Cycle 2	Cycle 3
	¥¥	ă	ASS	(1973)	(1861)	(1943)	Revise (*EIA Worksheet)	Revise :	Review (*CODSIA)
רגא	×								×
Magnette .		×							×
Manuel Int Car		>	,			>			,
McDonnell Deuglas	-	×	×				×	×	×
M.C.				×	×	•	,		
Hoterela MR 1 be		×			×		×		×
Not three	×								×
041		×		,					×
PARI P/M Comm		×		×		×	*	×	×
PRC		×							1
Rand Corp			1	×	1				;
Reydoon	×		×	×	××				××
A octor			×						×
ROLM					×	×			
				×	×	,			
Sanders		×				×	×	×	×
205	1	1	×	×	×:		×	×	×
Singer	×	×		×>	×>	,	*	£	.
Sellings AME				•	<	() (
Spery		×	×	×	×	×	×	×	×
Teledyne-Broom Telos				×	×	×××			
T. I.C. L. C.		Ī	,	•	Ĭ	×	***	,	
TRU		×	((×	×	: ×	×	k x	< ×
VIC.	×					×			×
V.110	1		k		,				×
Vought Corp	×				4				×
TOTAL				•1	Ħ	*	2	и	2

TABLE 1-2

OVERVIEW OF THE SDS PACKAGE EVOLUTION

Document Set Version	Reviewed	Ву
Review Cycle 1:		
April 15, 1982 Draft Review	Industry	DOD Components
Review Cycle 2:		
April 1983 Select Panel Review	Select Panel & EIA	JLC/CSM
July 30, 1983	EIA	JLC/CSM
August 31, 1983	EIA	JLC/CSM
Review Cycle 3:		
December 5, 1983 - Formal Review	Industry	DOD Components
May 1984 - Issue Resolutions	CODSIÁ	JLC/CSM
July 1984 - Issue Resolution *	CODSIA & EIA	JLC/CSM
October 1984 - Issue Resolution®	CODSIA	JLC/CSM, DOD
December 15, 1984 - Issue Resolution	CODSIA (Observer)	JLC/CSM
January 15, 1985 - Refinements*	CODSIA (Advisor)	JLC/CSM
January 30, 1985 - Refinements*	CODSIA (Advisor)	JLC/CSM
May 1985 - Refinements	-	DMSSO
June 4, 1985 - Formal Release	-	DMSSO .

NOTE: *Working drafts for incremental issue resolution. A limited number of working drafts were distributed for industry review.

the guidance provided by the EIA 13 major issues and the AIA 3 objections and 13 concerns. Due to the complexity of the issues and the approach used, little progress was made during the first rework cycle to address industry's concerns. To arrive at a standard acceptable to industry and to keep abreast of rapidly moving software technology required a different approach. The issue-driven approach was proposed by EIA and adopted by JLC/CSM during the select panel review meeting 24-26 May 1983 and subsequently used during review cycles 2 and 3.

Further, if the standard was to be developed in a timely, technology responsive manner, close cooperation between the JLC/CSM Subgroup, the CODSIA Task Group, and the SDS development contractor was mandatory. Such an iterative and cooperative resolution of issues was not implemented during the draft review cycle I and resulted in a failure to resolve critical issues identified by EIA and AIA. During subsequent review cycles, on assignment from the JLC/CSM Subgroup, DRC successfully recorded the essence of the Iterative negotiations between the Government and industry (CODSIA) representatives on the complex issues under discussion. As a result, during cycle 2 and 3, they were able to adjust the wording in the standard, DIDs and related documents to correspond with the agreements. This critical contribution to the process should be recognized for subsequent Revision A and future defense standards activities.

F. AN ISSUE DRIVEN REYEW APPROACH

The evolution of the SDS Package is based on an issue-Driven Approach (IDA) which was conceived by the EIA SDS review task group chairman, Ole Golubjatnikov from General Electric in June 1982 and applied during the EIA DOD-

5TD-SDS workshop³ in Dallas, Texas, September 20-24, 1982. The approach was subsequently adopted by both JLC^{4,5} and the select panel during the second review cycle in Wilmington, Mass during May 1983. During the third review cycle, IDA was further refined by CODSIA Task Group and JLC/CSM Subgroup. The adopted review approach is further described in Section 2 of this report.

The list of SDS issues evolved during the three review cycles as shown in Table 1-3.

TABLE 1-3
SUMMARY OF COMMENTS PER CYCLE AND ISSUE EVOLUTION

Review Cycle	Raw Comments		issues
	(Per Cycle)	New	Cumulative
Cycle I EIA Other	1370 } 5180	13 18 (AIA)	13 (EIA)
Cycle 2 EIA JLC Select Panel	76 8	11 18	24 42
Cycle 3 Industry DOD	2480 3401 } 5881	2	44 (Beginning of cycle) 35 (End of cycle)
TOTAL	11,829		

The initial set of 13 issues was identified by the EIA Dallas Workshop³ during cycle 1. In a similar vein, the AIA cover letter to JLC/CSM during cycle I review identified 18 general issues. These two lists had a high degree of commonality: the original list was expanded to 42 during the select panel review cycle 2. At the completion of CODSIA review cycle 3, a total of 55 issues had been identified and addressed. This list of issues is discussed in Section 3 of this report.

SECTION 2. THE ISSUE DRIVEN APPROACH (IDA)

A successful standard in an area of rapidly moving technology must be technically sound, adaptive to changes in technology, and broadly supported by a wide variety of developers and users.

DOD-STD-2167 is such a comprehensive standard, resulting from the joint efforts of the DoD and the defense industry. DOD-STD-SDS development process is based on IDA and represents a significant departure from the conventional approaches to defense standards development. The IDA addresses five fundamental issues inherent in defense standards development process:

- o A systematic approach to issue resolution in a complex and rapidly moving technology area.
- o Due process and consensus development.
- o Efficiency and effectiveness of voluntary standards process.
- o Active versus reactive standards development
- o Accelerated technology transitioning from R&D to battle operational environment.

This section will describe the IDA concept and summarize the lessons learned in applying IDA during the DOD-STD-2167 development process. The IDA concept can be further refined based on lessons learned and used as a model for the development of future standards in the MCCR area, such as the computer systems interface standards proposed by the CODSIA and the Defense Computer Resources Board (DCRB).

A. DEFENSE STANDARDS REQUIREMENTS

Properly conceived and useable standards are essential, both in the private sector and in defense, to cope with increasing technical complexities. Software development standards, in particular, are becoming important as a means to mitigate problems encountered in software development and the acquisition and operational use of computer based networks, systems and products. This observation is supported by the rapidly accelerating software standards development activities in the early 1980s with such national voluntary standards bodies as: EEF and ASTM, and international bodies, such as ISO and IEC.

As U.S. defense posture is critically dependent on software and resulting defense systems automation, it is imperative that the DoD maintain a national and international leadership position in software standards development which is a complex and extremely slow process. The adoption of national and international standards, while consistent with DoD policy on voluntary standards, would delay the introduction of modern software engineering practices in U.S. defense systems by 3-10 years. The development of modern software engineering practices and standards, and the resulting leadership in software engineering, is fundamental to the United States defense strategy based on the force multiplier concept and, therefore, must be aggressively pursued.

Beyond commercial software requirements, the defense software standards must also emphasize areas unique to the defense systems, such as security and phased acquisition life cycle of defense systems. At the same time, the defense

software standards should maintain compatibility in direction with national voluntary standards in areas such as commercial and reusable software and coding standards. Additional defense software requirements also exist internationally to assure compatibility with standards within the NATO defense alliance.

B. A SYSTEMATIC APPROACH

The IDA focuses on the root of the public review comments rather than fixing only the locally apparent problems. It considers conflicting goals, as well as alternative methods of solution starting from raw comments, to root concerns, to the basic issues. Once an issue is identified, it remains on the list, permitting traceability and follow-up to assure its continued resolution. The IDA makes the complex process of integrating the diverse and conflicting factors manageable through an incrementally evolving negotiation process. Further, this method provides a mechanism to assure currency with technology and changes in policy and business practice. The IDA is based on three major activities:

- Analysis of raw comments and bottom-up synthesis of concerns and issues.
- Top-down analysis and resolution of issues and correlation with other issues to resolve conflicts and assure overall compatibility.
- Review of detailed comments for each section and paragraph, and implementation of action items resulting from issue resolutions which is accomplished by rewriting affected sections and paragraphs of the standards and DIDs.

The five levels of the IDA structural concept is depicted in Figure 2-1: fundamental issues, issues, subissues, concerns and comments. The number of elements at all five levels at the end of CODSIA review cycle are summarized in the figure. As the IDA structure is fundamental to the approach, the individual levels are defined next.

I. FUNDAMENTAL ISSUES

These broad and pervasive issues are the primary MCCR life cycle cost and schedule drivers and are the basis for many industry's objections to any standard, including DOD-STD-SDS. They are:

- o Significant and unnecessary cost escalation (cost drivers)
- Unnecessarily constraining and restrictive
- o Isolation of software from the defense system and system engineering process
- Excessive data requirements

Generally little corrective action can be taken at the fundamental issues level, as these issues are too broad and pervasive. The fundamental issues are mapped into a number of different issues at the next lower level.

2. ISSUES

Issues are manageable areas of dispute, concern or controversy as grouped by life cycle, technology, methodology or project management considerations. They

SDS DEVELOPMENT APPROACH

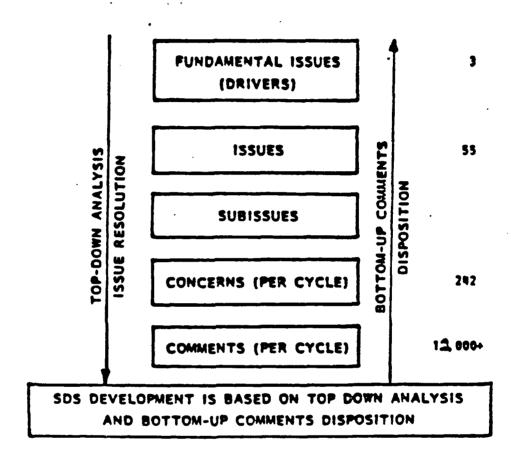


Figure 2-1. SDS Development Approach

frequently correspond to general or essential comments received during the SDS Package review. A total of 35 issues were identified during the three CODSIA review cycles. These issues are discussed in Section 3 of this report. Most complex SDS issues cannot be resolved at the Issue level but must be further subdivided into subissues for their resolution. Issues represent a permanent structure and are continued through the different SDS review cycles to verify their closure or to provide for future technology insertion and changes.

3. SUBISSUES

Subissues are the logical substructure of the more complex issues.

4. CONCERNS

Concerns are collections of related sets of detailed SDS review comments received for each review cycle. They exist only for the duration of the specific review cycle. They are frequently related to a specific SDS section and paragraph. Concerns may be used for making implementation changes to a specific SDS paragraph or mapped into the permanent structure of subissues, where they become part of an issue which may relate to a large number of paragraphs cutting across different standards and DIDs.

3. COMMENTS

Comments are the detailed raw comments received during the SDS review process. They are usually referenced against a specific paragraph of the standard or DID. Frequently, comments are only symptoms of the more basic issues which cut across the standard(s) and the DID(s).

C. DUE PROCESS AND CONSENSUS DEVELOPMENT

The development of complex defense standards in areas of rapidly moving technology is a significant technical and management challenge. The standard must not only be technically correct and dynamic but must also be broadly accepted and supported across a wide spectrum of defense applications and functionally different viewpoints.

Defense standards are usually drafted by a single individual, a single contractor, or a single service or agency component. For example, the DOD-STD-2167 draft was developed by DRC and the original set of DIDs by TRW. Much less frequently the standard is developed by a DoD or industry working group or a joint DoD and industry working group. The writers frequently lack the broad spectrum of viewpoints and experiences necessary to draft a technically sound and broadly supported standard capable of implementation across a wide and diverse range of applications. (The early drafts of DOD-STD-2167 and the related DIDs are good examples.) It is questionable that any single individual or a single organization can produce an acceptable document in isolation.

The development of a draft defense standard is followed by the coordination review process. Industry and DoD comments received during the coordination review fall into two categories: general and detailed. The detailed comments are referenced against a specific section and paragraph, while general comments frequently have no paragraph level references.

The comments received are usually processed by the same individual or organization (having a single viewpoint) which drafted the standard initially, with no or limited independent checks or balances provided by the system. The specific comments are processed by rewriting the referenced paragraphs of the standard. General comments are usually too difficult for corrective action and are frequently discarded by the process as too broad or too difficult to resolve.

As was vividly demonstrated during the first review cycle of 5DS, the above described approach in the case of complex standards in a dynamic technology area can result in a failure of the system to address the substantative issues contained in the large number of conflicting and competing comments. It is in this area that the joint steering and negotiation process by JLC/CSM subgroup and CODSIA Task Group made a major contribution during the CODSIA review cycle. In identifying issues, resolving conflicts and developing broadly based solutions and thus providing guidance and steering to the SDS contractor developing the detailed implementation of the standard and the related DIDs, a significant improvement was achieved in the consensus development process.

The second major contribution to the SDS process was the informal adoption by JLC/CSM Subgroup and CODSIA Task Group the broad principles used by the ANSI standards development process to assure due process and industry and DoD consensus. These principles are summarized below:

o Due Process

- Everyone with Direct and Material Interest
- Right to Express a Viewpoint
- If Dissatisfied, to Appeal Any Point
- Equity and Fair Play

o Consensus

- Substantial Agreement
- More than a Simple Majority
- Not Necessarily Unanimous
- All Views and Objections Considered
- Concerned Effort Made Towards Their Resolution
- Formal Voting Evidence II Required

Other Considerations

- Conflicts Resolved With Other Related Defense Standards
- Avoid Proprietory and Product Bias

D. EFFICIENCY AND EFFECTIVENESS OF THE VOLUNTARY STANDARDS PROCESS

The following activities and principles reflect the "lessons learned" or have contributed to the improved efficiency and effectiveness of the DOD-STD-SDS development process:

1. SELECT PANEL REVEWS

Premature release of draft standards for general public review and coordination should be avoided. The use of joint industry and DoD Workshops or selected working groups to review the early drafts of the standard avoids a large number of unnecessary comments during public review and the related processing costs and time delays. Such working groups need to be carefully composed to represent a balanced cross section of the industry.

2. USE OF CONTRACTORS, FOR STANDARDS DEVELOPMENT

The use of full time contractors (e.g., DRC and TRW for DOD-STD-SDS) has significantly accelerated the process of developing complex defense standards in the voluntary ANSI-like process.

3. USE OF DOD AND INDUSTRY STEERING GROUP

The use of JLC/CSM and CODSIA Task Group as the technical negotiations and management steering group for the DOD-STD-SDS development has significantly improved the quality of the standard and the process of consensus development and due process.

4. USE OF INDUSTRY EXPERTS

The use of industry volunteer experts as specific issue coordinators and problem solution developers has significantly improved the quality of DOD-STD-SDS and provided the necessary technical support to the CODSIA Task Group. This technical support base should be further expanded to support the Revision A development effort.

E. ACTIVE VERSUS REACTIVE STANDARDS DEVELOPMENT

There are two basic approaches to standards developments active and reactive documentaion. Active standards are planned, resulting from forethought as to their need and content. An excellent example of active standards development is Mr. Ford's dual role as an inventor of a mechanical wagon and a developer and promoter of traffic standards for his horseless carriage.

Reactive standards result from a need for some controls after the new invention or product has been introduced. Current generation DoD software standards such as MIL-STD-1679 are a good example of reactive standards. Software, for a long time, has been considered more of an art than engineering science. Therefore, software practitioners frequently object to attempts to establish software standards as excessive constraints to their intellectual and creative process. As software practice matures, and its economic and public safety impacts expand, the demand for software standards is escalating in both public and private sectors.

As a result of DOD-STD-SDS development, a fundamental change in defense software standards development has occurred. The defense industry has moved from reactive standards development in 1979 (Monterey I) to active standards development in 1985 (Revision A). As is evidenced by the issues addressed in the Revision A effort, such an artificial intelligence/expert systems (AI/ES), we are not only correcting past problems, but are also beginning to plan standards for advanced technology practice. Tertainly, it is much less costly to plan and implement standards along with the creation of a new product, process or

technology than to have to write standards later to solve problems created by the new introduction.

There are two basic positions that defense industry can take with respect to software standards developments active and reactive participation. Traditionally, most defense standards are the result of developments by DoD personnel. The defense industry generally participates only in a reactive review and comment mode. The current generation DoD software development standards, such as MIL-STD-1679 and MIL-STD-1644 were developed in this manner.

A wide variety of software categories, software development practices, and participant's viewpoints must be accommodated. No single individual or organization has all the required insights to develop an acceptable draft document. Close and active DoD and industry participation is absolutely mandatory if standards satisfactory to DoD user and industry developer are to be produced and kept current with changes in technology and business practice. Without active industry participation, the software technology is moving faster than the rate at which mutually acceptable standards can be developed, coordinated, and approved by the DoD alone.

The development of the DOD-STD-SDS Package over the last 6 years, as documented in this report, is an outstanding example of such DoD and defense industry active participation and cooperation. The initial release of the DOD-STD-2167 Package is not a perfect document, as is evidenced by the number of open issues for Revision A. At the same time, a review of other national and international software standards development projects clearly indicates DOD-STD-2167 to be a better and more cohesive standard than any other set of standards currently available or in development.

DOD-STD-2167 Package establishes the basic foundation for next generation defense software standards. This foundation will be improved by revisions based on technology evolution and implementation experiences with the initial DOD-STD-2167. The DOD-STD-2167 standards foundation will also be expanded as the other DoD software initiatives: Ads, STARS, SEI, and DARPA Strategic Computing Program become more closely coupled with the JLC SDS software initiative. Many of the unresolved longer term issues identified during the DOD-STD-2167 development will be resolved by products and R&D activities resulting from the other DoD software initiatives.

F. TECHNOLOGY TRANSITIONING FROM RAD TO BATTLE OFERATIONAL ENVIRONMENT

CODSIA Task Group 13-32 report to USD(R&E) entitled DeD Management of Mission - Critical Computer Resources observes that technology transitioning from R&D to battle environment is exceedingly slow and a major issue in MCCR management. Timely introduction of standards and their evolution with technology is a primary vehicle for assuring technology leadership and operational effectiveness in the battle environment.

The IDA, as demonstrated by DOD-STD-2167 development, provides the mechanisms for evolving defense standards as a function of technology developed by the public and private sectors. Issues such as AI/ES (Issues 36 through 60) combined with SDS revision process and implementation in the field provide a

334

mechanism for technology transitioning from early guidance to preferred practice and finally to enforced standards.

Many of the Issues Identified for DOD-STD-2167 Revision A are Identical to the Issues being addressed longer term by the DOD software Initiatives Ada, STARS and SEL. The coupling between these initiatives and the JLC software Initiatives for initial release of DOD-STD-2167 was minimal due to the early stages of these programs. As the STARS Program and the SEI becomes operational, considerable additional coupling and use of their R&D products is expected for Revision A and subsequent revisions of DOD-STD-2167.

SECTION 3. SDS DEVELOPMENT ESUES

The SDS Package is the product of an extensive public review and participation process which required the resolution of a large number of complex and conflicting factors. This resolution process and the detailed rationale used for the resolution of the 55 specific issues identified during the three SDS Package review cycles is documented in CODSIA Task Group 21-83 Report on the DOD-STD-2167 (SDS) Package Coordination Review dated November 1985.

Not all of the issues identified during the SDS Package evolution were fully resolved at the initial release of DOD-STD-2167. A number of issues were resolved based only on interim solutions. Revision A work is continuing for their full resolution, as well as a validation that the issues considered closed at the point of initial release of the standards package are actually closed based on field feedback.

The purpose of this section is to identify the 35 issues identified during SDS package evolution and provide status as to their priority and resolution. The 35 issues are identified in Table 3-1.

A. ESUE STATUS AT THE BEGINNING OF COOSIA REYEW (CYCLE 3)

Analysis of the industry review comments at the beginning of review cycle 3 revealed several dominant trends. In addition to making constructive comments, the participating companies expressed their positive reaction to the successive drafts of the SDS Package. Some respondents observed that the draft was already technically superior to the various existing standards being imposed.

In the area of constructive criticism, one dominant thread was the conviction that the standards would result in a significant and unnecessary cost escalation if released in their December 23 form. Although a serious and fundamental issue, the CODSIA task group decided these concerns were clearly identified and remedial action could be taken during the review process. Toward this end, the group identified eight primary issues and nine secondary issues for resolution. The cost drivers were included in the list of primary issues.

The eight primary issues and their tracking numbers are listed below in a priority order:

- 1. Issue 25: Tailoring
- 2. Issue 41: Software Development File (Allas Folder)
- 3. Issues 16, 17, 12, 21: Informal Testing
- 4. Issue 7: Ada
- 5. Issue 8: Firmware
- 5. Issue 6: Systems Interface and Isolation of Software
- 7. Issue 43: Automation
- 8. Issue 27: Revision Strategy

Although significant changes were incorporated in the SDS Package (dated 5 December 83) to resolve comments about SDS being too constraining and restrictive



SUMMARY OF FINAL STATUS OF ISSUES - CODIA REVIEW CYCLE (6 JUNE 1983)

2 2		KCK		_	×																					=							
ox d-Valle	Policy	& >108		X(2168)			×		X			i	x(43)													X(2168)							
Partially Closed-Validate	2	HOBK	×						×		×								ł								;	×	×				
4		X A		×	×				×	×	×		×													×	×						
į	(Interim	Keleaki																		•													
Tax of D	מניל ל	COM				×						×		×	×	×	×	×	×	×	×	×	×	×	×					×	×		
	CODSIA	Friority	U	S	⊢	~			Q.	۵.	۵.	&	R(6)	U	U	s,	-	U	R(17)	۵.	R(17)	ပ	U	R(17)	U		U	٩	U	P (\$)	U	U	
		Issue Description	6 Additional DIDs	SDS/SQS Relationships	Supportability	Too constraining	Evolutionary acquisition T	Systems engineering	versus soliware	Ada.	Firmera .	SOS Imp. strategy	MIL-STD-499 relationship	Architecture definition	Levels of control	Design methodology	MMI coverage	Phasing of LC activities	Test description	Informal testing	Formal testing	Product baseline	DIDs nef policy/standards	Test & integration	STS & test documentation	Quality assurance	Coding standards	Telloring approach	Commercial/reusable SW	SOS revision strategy	Scope/application	SDS Organization	
	2	Š	-	~	_	•	~			7	-	•	2	=	~	=	=	~	2	-	=	<u> </u>	22	7.	22	2	:	25	36	2	=	52	

SUMMARY OF FINAL STATUS OF ISSUES - CODIA REVIEW CYCLE (6 JUNE 1983)

			i	(2	rtially Cl	Partially Cloxed-Validate	21
Issue Description		CODSIA	Closed (JLC/ CSM)	Open (Interim Release)	Rev A	HOBK	Other Policy & Stds	Joint
Program support library	library	U		×				
8		U		×				
Formal quality review	ivhe.	U		×				
Deseile managem	management concept	U						
Specmanship/relation	- Wo	U						
	detail	U						
		U						
Additional DIDs phasi	hasing							
Subcontract appling?		U						
Design architectur	•	U						
Unit development folder	folder							
Security		U						
Automation		•						
Fragmentation (m	station (regent plans)	so.				,		
Adequacy of technical	nical conten	=						
DIDs to be super	papa	\$						
Training		ž						
DIDs collapsing		'n						
Capacity miety margina	and Share	U	}	×				
Editorial	,	1		×				
Unclear		-		×				
	ceabillty	U		×				
	8	υ		×				
MIL-STD-490 B1/C	Ū	R(6)						
Excessive data		۵.						
TOTALS			z	0	91	91		7
			!	,))	,	,

338

KEY

(Issue 4), this fundamental issue was still a dominant concern among industry reviewers who characterized the package as having too much how-to direction, and verging on micromanagement in other areas. However, this fundamental issue was considered too general for corrective action and was mapped into three specific primary issues:

- o Issue 42: Software Development Folders
- o Issue 17: Informal Testing Constraints
- o Issue 7: Ada® Sultability/Compatibility

By successfully addressing these specific issues, the fundamental issue of being too restrictive would be substantially resolved.

Some respondents noted the trend to isolate the software development and acquisition methodology and terminology from that of general systems engineering and system acquisition. The structure of the SDS activities (e.g., absence of requirements generation methodology) and associated policies are perceived as exacerbating this situation. The Task Group decided to address this fundamental issue, which requires significant work, in a future update to the SDS Package. The decision to postpone problem solution to a future update merits some elaboration.

The CODSIA Task Group recognized two classes of action recommendations:

- 1) Short term action temporary solutions to problems requiring extensive technical work for incorporation in the initial release of the SDS Package (4 June 1985).
- 2) Long term action the final solutions to problems which require further technical work, and which are expected to be implemented in subsequent revisions of the SDS Package.

The Task Group created this distinction because of pragmatic considerations relating to the need for early release of the SDS Package versus the time and effort required to research the changes. Therefore, a revision to the Interim Version is essential, and the Revised Version should be implemented within 2 years (i.e., June 1987). It should also be noted that these deferred issues have not been more effectively addressed in any existing software development standard, so the new standard is not a regression.

The priority of the 44 issues identified by the beginning of review cycle 3 was established by the CODSIA Task Group based on:

- o The impact of the issue on industry practice.
- The assessment of the issue status as to its resolution based on industry comments received and Task Group review of the SDS Package.

The relative priority of the issues, based on the December 1983 draft, was categorized as:

o Primary (3 issues)

^{*} Ada is a registered tracemark of the U.S. Government Ada Joint Program Office.

- o Secondary (10 issues)
- o Tertiary (9 issues)
- Closed (13 issues)
- Remapped (4 isues)).

Issue 4 (Too constraining/restrictive) was considered a fundamental issue and excessively broad for corrective action and was remapped into issues number 7, 16, 17, 18, 21 and 41.

Issues 16, 17, 18 and 21 addressing test related issues were collected under a single issue 16.

B. ISSUE STATUS AT THE POINT OF SIX PACKAGE INITIAL RELEASE.

The status of issues at the point of SDS Package initial release is summarized in Table 3-1. The status table includes 42 issues identified during the two previous review cycles. Issues 43 through 55 were added during review cycle 3. The table identifies the CODSIA priority of all issues at the point of SDS Package initial releases primary, secondary, tertiary, and closed. The closed issues are subject to validation based on field usage feedback. Partially closed issues indicate the proposed resolution strategys Revision A, MIL-HDBK-287, changes to other policies and standards and revisions to Joint Regulation.

C. LIMITED COCRDINATION AND PROPOSED CHANGES

Three sections were added to the standard during December 1984 with essentially no industry review because these section were created at the final stage of the review process, when time did not permit their wide circulation. The three sections in question are:

- o 3.8 Software Quality Evaluation
- 5.9.1.5 Risk Management
- o Appendix D Tailoring Guidelines

The first two were prepared by the Government contractor (DRC). The CODSIA Task Group objected to their inclusion (particularly 3.3 which is voluminous) without adequate reviews. Those objections were overruled however, and the sections were included in the standard. The Appendix D was prepared by Mr. O. Golubjatnikov, a member of the CODSIA team, but was only reviewed by the members of the team before being offered to the Government for inclusion. It is recommended that each of these sections be given particular scrutiny during upcoming reviews.

An Appendix E, "Application Guide and Example for Development of Prime Items and Critical Items that Contain Software and Firmware Components", was also prepared by the EIA members Messrs. O. Golubjatnikov and John C. Hamilin of the CODSIA team. This appendix specifically addressed the MIL-STD-490 B1/32 issue number 34 and the related isolation of software from the system engineering process. This appendix was voted and unanimously approved by EIA membership, but did not receive unanimous CODSIA Task Group approval and thus was not incorporated in the inital 4 June 1985 release of DOD-STD-2167.

SECTION 4. SDS IMPLEMENTATION AND PLANS FOR REVISION A

17.

The SDS Package was approved by DMSSO for DoD-wide usage 4 June 1985. The smooth transitioning from a wide variety of currently service-unique and single program software acquisition practices to the uniform DOD-STD-2167 practice and its successful implementation and evolution with technology advances in DoD and industry depends on:

- I. Training and tailored application of the standards package.
- 2. Field monitoring and implementation feedback to avoid misapplication of the initial version of the standards package.
- 3. Continued development of improved solutions for issues closed for the initial release based on interim solutions.
- 4. Continued evolution of the SDS Package through technology insertion from public and private developments.

A. GOVERNMENT DOD-STD-2167 IMPLEMENTATION PLANS

The JLC/CSM Subgroup and the Services have initiated a DOD-STD-2167 implementation program. The major elements are:

- o JLC/CSM Subgroup DOD-STD-2167 Implementation Concept/Strategy Plan
- Service-unique DOD-51D-2167 Implementation Plans
- o DOD-STD-2167 Training Courses at executive, management, and technical levels
- DOD-STD-2167 (MIL-HDBK-237) Handbook development
- DOD-STD-2167 "Heip" line (with level-of-effort contractor support)
- DOD-STD-2167 Implementation Feedback Survey of current users
- o DOD-STD-2167 Implementation Evaluation
- o Industry and Government Briefings and Tutorials
- o JLC Workshops (Orlando II)

As an example of the effort applied, during 1984/1985, the JLC/CSM Subgroup alone has given over 25 briefings and 5 tutorials to industry and Government on the technical aspects of DOD-STD-SDS package.

B. INDUSTRY DOD-STD-2167 EMPLEMENTATION INITIATIVES

To provide proper DOD-STD-2167 Introduction and assure its continued evolution with technology and field feedback, the CODSIA Task Group 21-83, the industry associations and the individual corporations are taking numerous actions. The following are examples of industry DOD-STD-2167 initiatives.

o Joint Industry Conferences and Tutorials. During 1984 and 1985 industry associations (AIA, EIA, NSIA) and professional societies (IEEE, ACM) have held numerous joint industry conferences focusing on different aspects of defense software. These conferences offer papers on DOD-STD-2167 and DOD-STD-2168. Some conferences have also offered 4-hour tutorials on DOD-STD-2167, DOD-

STD-2168, and Software Standardizaton Activities. Additionally, the DPMA has offered two-day tutorials on DOD-STD-2167 and DOD-STD-2168. EIA is also offering DOD-STD-2167 and DOD-STD-2168 tutorials during its annual EIA G33/34 workshops. These industry association initiatives are epected to continue.

o EIA Workshops on DOD-STD-2167 and DOD-STD-2168 Development. EIA conducted special panels on DOD-STD-2167 and DOD-STD-2168 development during its annual EIA G33/34 workshops in 1981, 1983, 1984, and 1985. These workshops are attended by both industry and Government personnel and complement the JLC-sponsored (Monterey I, II, and Orlando I) workshops on DOD-STD-SDS development.

These workshops provide an opportunity for Government, industry, and academic personnel to address specific DOD-STD-SDS related problems and issues and participate in the DOD-STD-SDS Package development process. The EIA workshop panels are led by CODSIA Task Group members, industry association reviewers, Government DOD-STD-SDS contractor and JLC/CSM Subgroup personnel and provide an excellent forum for resolution of problems and development of recommendations to the software standard. These annual workshops are expected to continue.

- o Industry Upgrade of In-House Software Standards and Procedures. A number of companies have already adopted, on a voluntary casis, DOD-STD-2167 as their in-house software development standard. These companies have acted as DOD-STD-2167 test beds and provide useful feedback to the SDS Package development during the second and third review cycles. More recently, other companies have begun to upgrade their in-house standards and procedures to assure compatibility with DOD-STD-2167 requirements for contractual compliance.
- o Industry Implementation of DOD-STD-2167 Environments and Automation Tools. Tri-service standardization on a single software development standard with a consistent set of DIDs creates an environment conducive to investments in software development automation. A number of companies have ongoing efforts to automate the generation of DOD-STD-SDS required products. It is expected that such environments and automated tools will not only be operated intermally by the major system houses, but will also be offered, as products by houses specializing in marketing software tools and environments.

C. ASSESSMENT OF DOD-STD-2167 INTERIM RELEASE

The early applications of the SDS Package on Government proposals and contracts provides an opportunity for a detailed assessment of the package, as well as a detailed validation of proper implementation of the issue resolutions. Both JLC/CSM Subgroup and CODSIA are planning to establish a joint data collection mechanisms so that the feedback from early applications can be promptly evaluated and the required corrective actions initiated during the Revision A cycle.

The implementation of joint JLC and CODSIA issue resolution agreements during the coordination review required a large number of changes. The language for these changes was largely implemented by the Government contractor, DRC and reviewed by JLC. The development of the last two versions of the documents set were driven by DMSSO requested changes and had no CODSIA participation.

The development of the three document set versions prior to the last two versions had only limited CODSIA review because CODSIA was acting only in an

advisory capacity. These reviews were further constrained by tight release schedules. Therefore, with the public release of the documents, it is essential that a detailed review be conducted to validate the issues and concerns that have been agreed to by JLC and CODSIA. This document provides a baseline on the status of the specific issue and concerns.

One corporation conducted a detailed review of the January 30, 1983 version of the document and concluded that several of the Issues considered closed by JLC and CODSIA task group are, in fact, not yet fully closed. The specific Issues involved (16, 41, 43) are prioritized as "resolved in principal but require refinement." Such reviews should be continued so that the necessary corrective action can be taken through change notices or Revision A to the standard.

D. SDS REVISION A OVERVEW AND MILESTONES

At the point of initial release of DOD-STD-2167, a number of issues were resolved on an interim basis only subject to further R&D. This section describes the initial CODSIA plans for the Revison A as coordinated with 3LC/CSM Subgroup. The SDS revision process is envisioned to be an ongoing activity leading to future revisions.

Revision Process Drivers. The development of SDS Revision A is driven by the following five major sources of activities:

- o Open issues from the coordination review (cycle 3)
- Feedback from early field usage of SDS Package
- o STARS program and Software Engineering Institute (SEI) research and development activities
- Ada technology and practice evolution
- Evolution of software engineering technology and industry practice.

The Revision A process provides an interface to the above five major source of activities and converts them to the evolving set of SDS Issues. The Revision A process consists of the following four major activities:

- o identification of the issues
- Analysis and resolution of the issues
- Draft Revision A development and coordination
- Revision A implementation and field feedback

Revision A Milestones. The initial set of major milestones for the overall Revision A process are:

Identification of Issues

•	Initial Set of Issues	May \$3
•	Formal Coordination lasses	Jan-Oct 84
•	Rev A Issues Baseline	Sept 15
•	Feedback from Field Use	June-85 - Mar 86
•	CODSIA Issues Paper	Oct \$5
•	STARS Program and SEL Regularments Issue	October 95

Definition and Coordination

- Analysis of Issues
 - o Proposed Resolution of Issues Received from Jul-Dec 85
 CODSIA Focal Points and HQ AFSC Staff
 o Analysis SOW Prepared Sept 85
 o Analysis/Revision Contract Awarded Oct 85
 - o EIA St. Louis Issues Workshop 16-20 Sep 85 o Analysis Completed Mar 26
- o Coordination/Implementation of Revision A
 - o Preliminary Draft Jun. 36
 o Review/Coordination Dec 36
 o Implementation June 37

Revision A Objectives and Goals. The specific goals of the Revision A process are:

- o To validate the detailed implementation of Issues considered to have been closed in the initial release of the SDS Package.
- o To close off the open issues remaining from the coordination review cycle. A number of issues are partially closed while other open issues have been resolved based on interim solutions only.
- To provide feedback from early field usage of the SDS Package and implement corrective action through change notices, Revision A and Handbook changes.
- To incorporate early R&D products from the STARS program and the SEI. Provide STARS program direction for SDS areas requiring R&D.

In addition to the above specific goals, the Revision A process is guided by the following broad objectives:

- Provide technological currency of SDS
- Accelerate software technology transition
- o Improve software portability
- Encourage software productivity and automation
- Support Ada Introduction
- o Encourage production of quality software
- o Encourage software reuse
- Improve post-deployment support and reduce life-cycle costs
- Provide flexibility for developer innovations
- Minimize constraints of acquisiton process on contractor's internal processes while enforcing sound discipline.

E. SUMMARY OF OPEN ISSUES

During the SDS Package evolution, a total of 55 issues were identified. At the completion of the coordination review cycle, 29 of these issues are closed while 18 are open. Practically all of the open issues are partially resolved or have been resolved based on interim solutions. During the joint JLC/CSM and COSDIA Task Group meeting on June 7, 1985 the following summary status and CODSIA reprioritization of open issues was documented:

0	Primary Issues Requiring R&D	•
0	Primary Issues - Resolved in principle but	
	require refinement	3
•	Other Primary Issues	2
0	Secondary Issues	•
•	Tertiary Isues	_5
•	Total Open Issues	18
0	Resolved Issues	29
0	Issues Requiring Govt/Ind Action	
0	Remapped Issues	_
0	Grand Total	55

The following four primary issues (including 3 issues consolidated into system engineering) require considerable R&D efforts

- o Issues 6,10,29,54: System Engineering
- o Issue 7: Ada Compatibility
 - Coding Standard
 - DID Tailoring
- o Issue 8: Firmware

(

o Issue 25: Tailoring Appendix

The following two new primary issues are considered opens

- o Issue 54: MIL-STD-490 B1/C1 (SDS compatibility revision)
- o lissue 55: Excessive data

The following three primary issues are completely resolved in principle, but require considerable refinement:

- o Issues 16,17,13,21: Informal Testing
- o Issue 41: SCF Group
- o Issue 43: SDS Encourage Automation

The following four primary issues relate to the SDS development process and not the SDS product. These issues are considered closed as long as the planned SDS development process is moving forwards

- o Issue 91 Implementation
- o Issue 27: Revision Strategy
- lssue 46: DIDs to be Superceded
- o Issue 47: Training

The following four open issues are categorized as secondary:

o Issue 2: Relationship to 2163 (Rewrite 5.3)

o Issue 13: New Methodologies

o Issue 44: Fragmentation of Mgmt Plans

o Issue 48: DID Collapsing

The following five open issues are categorized as tertiary:

o Issue 3: Supportability

o Issue 5: Evolutionary Acquisition

o Issue 14: SDS Discussion of Personnel Subsystem

o Issue 50: Editorial

o Issue 31: Unclear

F. NEW REVISION A INITIATED ISSUES

As a result of EIA Computer Resources Workshop⁷ on DOD-STD-2167 conducted in St. Louis, MO., 16-20 September, 1985, an approach to Artificial Intelligence/Expert Systems (AI/ES) in the DOD-STD-2167 acquisition environment is recommended. The approach proposed by the EIA workshop is to handle AI/ES as a new category of software within the DOD-STD-2167 tailoring concept. To address this proposed new category of software five new issues are proposed.

- 1. A1/E5 Technical Development Methodologies are inconsistent with DOD-STD-2167 (Issue 56).
- 2. AI/ES Life Cycle Varies from Traditional (Issue 57).
- DOD-STD-2167 Documentation is Insufficient for AI/ES Systems (Issue 58).
- New AI/ES Optimized Life Cycle Management Methods are required (Issue 59).
- 5. Other A1/E3 Unique Issues (Issue 60).

No AI/ES comments or concerns were received during the three DOD-STD-SDS review cycles. Based on lack of comments, it was felt that the AI/ES technology practice was not sufficiently mature to initiate guidance or standardization or that the volume of business is insufficient to be of concern for DOD-STD-2167 initial release.

This assessment was changed as a result of the excellent work done by panel 2 of the EIA Computer Resources Workshop In St. Louis under the co-chairmanship of Messrs, R.M. Bond of ARINC, G. Wigle of Boeing Aerospace and D. Preston of ITTRI.

The early conclusions of the workshop panel 2 are as follows:

- o 2167 is tallorable for AI/ES
- o AI/ES has potentially serious impacts on DOD-STD-2167 documentation

The primary drivers for AI/ES incompatibilities with the initial release of DOD-STD-2167 are as follows:

- Al Development Methodologies
 - Exploratory Programming

- Bottom-up Non-Hierarchical
- Knowledge Engineering not Addressed by DOD-STD-2167
- Al Applied to Fuzzy Problems
- Executable Data/Self-Modifying Systems

SECTION 5. SUMMARY

This section provides a summary of the paper including assessment of the SDS Package, acknowledgements, conclusions and recommendations.

A. ASSESSMENT OF SDS PACKAGE

The release of DOD-STD-2167 to DoD-wide usage represents a significant accomplishment. Most of the objectives and goals set for the DOD-STD-2167 by the DoD and industry have been met. Work is continuing to improve the standard where issues are still outstanding or where technology is driving future changes. Field experience within the DoD and defense industry and voluntary usage outside the DoD will provide the final evidence of its success.

To provide a more detailed assessment of the DOD-STD-2167 (initial release), the following criteria are applied:

- 1. JLC SDS objectives
- 2. JLC/CODSIA issues criteria
- 3. EIA/AIA White Paper criteria 10
- 4. DODD \$120.21 Acquisition Streamlining Directive criteria 11
- 5. General Standards Value criteria 12

I. JLC SDS OBJECTIVES

JLC 5DS objectives are summarized below:

Produce a complete, consistent tri-service set of acquisition, development and support standards which:

- Establish a well-defined and easily understood software acquisition and development process
- Provide adequate visibility during software development and acquisition
- Reduce confusion and eliminate conflicts in existing standards
- Are compatible with modern methods of developing software
- Provide cost benefits over the entire life cycle
- o Increase probability of obtaining quality software

The first three objectives are completed with the initial release of the standard. The full attainment of the last three objectives are subject to SDS implementation, Revision A and the assessment of field feedback from early applications.

2. JLC/CODSIA BSUES CRITERIA

The assessment of the initial release against the JLC/CODSIA issues criteria is summarized in Section 4. All 55 issues have been closed or have interim solutions contained in the 4 June 1985 SDS package. Eighteen issues are still open

for refinements and improvements during the Revision A process. Four issues require continued action by JLC/CSM Subgroup during DOD-STD-2167 implementation:

- o Issue 9: SDS Implementation
- o Issue 27: Revision strategy
- o Issue 46: DIDs to be superceded
- o Issue 47: Training

All four actions have been initiated, and as long as they are continuing, they are considered closed.

3. EIA/AIA WHITE PAPER CRITERIA

The EIA/AIA White Paper criteria are summarized below:

- o Sound Discipline Without Inhibiting Effective Design
- Flexible Standard to Accommodate Software of Differing Scope and Applications
- O Development and Management Methodology Must Accommodate Continuing Technology Advances Without Loss of Discipline
- Provide Clear Definition of Post-Delivery Support Requirements
- Careful Integration of Diverse and Conflicting Factors

Each of the above criteria has a number of sub-criteria. A review of the subcriteria indicates that all of them have been mapped into the 55 JLC/CSM issues and that all of these are closed or have action items planned during Revision A.

4. DODD 4120.21 ACQUISITION STREAMLINING DIRECTIVE CRITERIA

The nine criteria contained in DODD \$120.21 directive ¹¹ which apply to DOD-STD-2167 are listed in Table 1-4. The initial release of the standard is fully responsive to these criteria, with activities continuing during implementation and Revision A phases.

5. Standards Value Criteria

S. C.

The broadly quoted standards value criteria 12 is listed below:

- o Standards should Educate
- Standards should Simplify
- o Standards should Conserve
- o Standards are a base to Certify Against

The DOD-STD-2167' and its implementation plans are responsive to all four criteria listed above.

TABLE 5-1

STREAMLINING	INITIATIVE	CRITERIA	(DODD	A120 211
21050000111111	114111V 14 E	CRILERIA	11 14 21 21 2	8 1 2U.ZIJ

			Criteria	Supported By	
_	Initiative Criteria	JLC SDS Policy	SDS Package	SDS Implementation	Revision A
1. 2.	System-Level Functional Requirements Cut Off Referenced	N/A	х	×	×
3.	Documents	×	*	N/A	N/A
	Baseline	N/A	x	x (STARS	x
4. 5.	Stds & DIDs	×	×	×	×
6.	Cost Management	N/A	x	x	N/A
7.	not "How To"	N/A	×	N/A	×
_	DIDs Only	×	2	N/A	N/A
1.	DIDs Consistent With Task Requirements	N/A	×	×	· N/A
9.	Only Required Data Ordered	N/A	N/A	¥	N/A

Notes: N/A - Not Applicable; x - Criteria Satisfied

B. ACKNOWLEDGEMENTS

The roots of the SDS Package originate in the mid 1970s. The initiation of its development in 1979 by JLC/CRM represents considerable vision and executive level commitment. Credit is due to the past and present JLC/CRM chairmen BGen. Donald Lasher, Col. John Marciniak, Capt. Dave Boslaugh, and Col. Harold J. Archibald.

The development of the SDS Package represents a significant accomplishment by the more than 300 individuals and over 150 corporations and Government organizations participating in its development. DOD-STD-2167 will have a significant impact on the \$10 billion of software being developed in 1935 for the currently installed MCCR base of 185,000 computers. This impact will increase rapidly as the development of defense systems software triples by the end of the decade and the industry phases over to DOD-STD-2167 practice.

The quality of the SDS Package and its continued evolution is the direct result of the IDA adopted by the JLC/SDM Subgroup chairman Capt. Lee Cooper. The contributions made by Capt. Cooper in the establishment of a framework of cooperation between the DoD and industry are absolutely critical to the successful results produced and the continuing evolution of the standards package.

Further acknowledgements are due to:

o JLC/CSM Subgroup members and past and present Chairmen Lt. Col. Oberkrom, Lt. Col. Casper Klucas, Major Larry Fry, Lt. Cdr. Mike Gabt and Capt. Ltd Copper.

- JLC and EIA workshop participants and their sponsoring organizations.
- o Industry and DOD reviewers and their organizational sponsors.
- o Industry issue coordinators, special working groups and their organizational sponsors.
- o SDS Package development contractors DRC, TRW and Logicon and in particular Mr. Dave Maibor of DRC.
- o CODSIA Task Group 21-83 chaired by Mr. Jim Heil of ITT and their organizational sponsors.
- EIA G34 Computer Resources Committee chaired by Mr. Jerry Raveling from Sperry.

The material presented in this report has been extracted from the CODSIA Task Group Report 21-83 on the DOD-STD-2167(SDS) Package Coordination Review with full credit due to the members of the Task Group.

Author's participation in the SDS Package development represents a significant investment by General Electric in the voluntary standards process. In particular, the resource commitment provided over the years by Messrs. F. M. DeBritz and C. B. Clarkson was critical in the formulation of the IDA.

C. CONCLUSIONS

(1975)

 $\mathcal{F}_{\mathcal{F}}$

The most significant conclusions related to the SDS product are:

- 1. The quality of the SDS Package, as measured by issues resolved, is directly related to the voluntary effort put forth by industry and the DoD.
- 2. The SDS Package is a significant accomplishment and meets most of the criteria established by the DoD and industry:
 - o JLC SDS objectives
 - o JLC/DODSIA issues criteria
 - o EIA/AIA white paper criteria
 - DODD 4120.2 Acquisition Streamlining Directive criteria
 - General standards value criteria

Further, work is continuing for improvements against the above listed criteria where not yet fully met.

- 3. SDS Package provides a standards foundation for technology insertion from the other DoD software initiatives Ada, STARS and SEI, as well as the private sector technology developments.
- The accomplishment of a single software development standard is not without risks. The range of computer programs to be covered by DOD-STD-2167 is extremely broad. Tailoring of the standard is absolutely essential if the flexibility for spanning the wide range of defense systems and the variations in project size and software categories is to be achieved.

The most significant conclusions related to the SDS Package development process are:

- The IDA represents a significant change from the conventional defense standards development process and was critical to the quality of the DOD-STD-2167 and its acceptance by the industry.
- 2. The IDA can serve as a model for the development of future standards in the MCCR area.
- JLC/CRM commitment to the Revision A process and SDS implementation plans was essential for industry endorsement of the initial release of DOD-STD-2167.

D. RECOMMENDATIONS

The following are the most significant recommendations:

- I. Industry and DOD should provide adequate resources to complete the planned Revision A process by June 1987. The issue resolutions should be completed by June 1986.
- 2. Industry and DoD volunteers should establish JLC/CODSIA coordinated working groups to address each of the Revision A open issues.
- Industry associations and DoD organizations should consider sponsoring JLC/CODSIA coordinated joint DoD and industry workshops to address the following seven major Revision A issues:
 - 6 Automation
 - o Methodology
 - o Reusable Software
 - o Ada
 - System engineering
 - o AL/ES
 - o Firmware
- Industry and DOD should refine the IDA for Revision A and use it as a model for the development of future standards in the MCCR area.
- 5. The coordination and technology transfer between the JLC SDS software initiative and the other DoD software initiatives: Ada, STARS and SEI should be improved.
- 6. DMSSO should consider developing data bases, tools and network access for the automated processing of public review comments.
- 7. Industry and DoD should support the proposed DOD-STD-2167 implementation plans so that the full benefits of the SDS Package can be achieved in a timely manner.
- 8. Individuals interested in participating in working groups and organizations considering sponsoring DOD-STD-2167 Revision A issue resolution activities should contact the following:

DOD: Capt. Rick Butler or

Capt. Lee Cooper

Andrews AFB, MD 20334

(301) 981-5731/4 AV 858-5731/4 AIA: Mr. Austin Maher

Singer Kearfott Corp. 150 Totowa Rd Wayne, NJ 07470

(201) 785-6607

EIA: Mr. Ole Golubjatnikov

General Electric Co. FRP 1, Room D6 Syracuse, NY (315) 456-4744 NSIA

Mr. Jim Heil ITT Avionics 100 Kingsland Road Clifton, NJ 07041 (201) 284-2946

REFERENCES

(2

- CODSIA Task Group 21-83, Report on the DOD-STD-2167 (SDS) Package Coordination Review, Washington, DC, November 1985.
- CODSIA Task Group 13-82, Report to USD(R&E) on DOD Management of Mission-Critical Computer Resources, Volume I and II Washington, DC, March 1984.
- 3. EIA 1982 Workshop Panel 1 Report, "Impact of Tri-Service Initiatives on Software Development (Draft MIL-STD-SDS Coordination)," EIA, Washington, DC, September 1982.
- 4. CAPT Lee Cooper, "Issue Resolution Procedure," JLC/CSM, Andrews AFB, 21 July 1983.
- 5. CAPT Lee Cooper, "Discussion of Proposed MIL-STD-SDS Issue Resolution Strategy," JLC/CSM, Andrews AFB, 21 July 1983.
- 6. David S. Maibor, "Summary of Concerns Raised for Each Issue," DRC, Wilmington, MA, Original Issue 2 November 1984, as updated by Capt. Cooper 7 June 1985.
- 7. DOD/CSSD, STARS Software Technology for Adaptable, Reliable Systems Defense Industry Briefling, San Diego, CA, 29 April 1985.
- Software Engineering Institute Industry Affiliates Symposiu, Carnegie-Mellon University, Pittsburgh, PA, 30 September 1985.
- Electronic Industries Association, Computer Resources Committee, 19th Annual Workshop Report, St. Louis, MO, 16-20 Septmeber 1985.
- 10. "Suggestions for DOD Management of Embedded Computer Software in an Environment of Rapidly Moving Technology," EIA and AIA, Washington, DC, March 1982.
- 11. Draft DODD \$120.21, "Acquisition Streamling," DOD, Washington, DC. June 1985 and associated attachments.
- 12. Charles D. Sullivan, Standards and Standardization, Marcel Dekker, Inc., New York, 1983.

CODSIA 21-83

DOD-STD-2167 REV A PLANS

O. GOLUBJATNIKOV

GENERAL ELECTRIC COMPANY

VICE-CHAIRMAN CODSIA 21-83 TASK GROUP

MARCH 1986

1.15

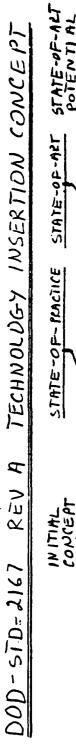
COVERAGE

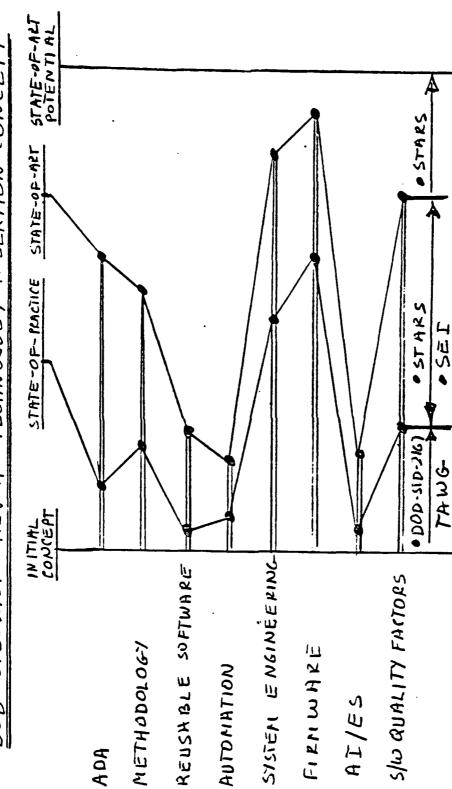
- REV A SCHEDULE
- TECHNICAL ADVISORY WORKING GROUPS (TAWG)
- TAWG GUIDELINES
- SUMMARY STATUS
 - ADA TAWG
 - AI/ES TAWG
 - FIRMWARE TAWG
 - SYSTEM ENGINEERING TAWG
 - REUSABLE SOFTWARE TAWG
 - EVOLUTIONARY ACQUISITION TAWG
 - SOFTWARE QUALITY FACTORS TAWG
 - METHODOLOGY TAWG
 - AUTOMATION TAWG



DOD-STD-2167 REVISION A SCHEDULE

•	REVISION A KICKOFF MEETING	15-16	JAN	1986
•	CODSIA 21-83 REPORT REVIEW & COMMENTS	JAN-	-MAR	1986
•	INITIAL INPUTS FOR REV A DRAFT	15	MAR	1986
•	CODSIA MEETING (CRYSTAL CITY)	20-21	MAR	1986
•	CODSIA/JLC MEETING (SAN DIEGO)	9-11	APR	1986
•	TAILORING WORKSHOP	15-16	APR	1986
•	FINAL INPUTS FOR REV A DRAFT	30	APR	1986
•	REV A RELEASED FOR COORDINATION	15	AUG	1986
•	60-DAY REVIEW COMPLETED	15	OCT	1986
•	REVISION A FINAL DRAFT REVIEW	1	JAN	1987
•	REVISION A TO DMSSO APPROVAL	1	MAR	1987
•	REVISION A RELEASED		JUL	1987





STATE-OF-ART STATE-OF-PRACTICE, DOD-STD-2167 THWG ASSISTS IN CLOSING THE REFLECTS ロミサ DOD - STD-2167 PRACTIC E

SEL

CODSIA 21-83 TECHNICAL ADVISORY WORKING GROUPS (TAMG) ON DOD-STD-2167

TANG	SPONSOR	CONTACT
ADA (SDSADAWG)	ACM SIGADA	D.G. FIRESMITH (MAGNAVOX)
AI/ES TAMG	EIA 6-34	D. PRESTON (IITRI)/6. WIGLE (BOEING)
FIRMWARE TAWG	AIA	A.J. MAHER (SINGER-KEARFOTT)
SYSTEMS ENGINEERING TANG	AIA	A. M ^C CULLOCH (HUGHES)
REUSABLE SOFTWARE TAWG	STARS/AS AREA	R.D. KOLACKI (SPAWAR)
EVOLUTIONARY ACQUISITION TAWG	NSIA/COMCAC	H. LYNESS (GE)/E.C. BAUDER (GTE)
SOFTWARE QUALITY FACTORS TANG	NSIA	J.H. HEIL (ITT)
METHODOLOGY TAWG	NEED SPONSOR	6. SUMRALL (USA)/H. LEHRHAUPT (GE)
AUTOMATION TAWG	NEED SPONSOR	

JLC/CSM SIBGROUP CAPT R. BUTLER

TAILORING WORKSHOP

MORKSHOP

STARS INTERFACE WORKING GROUP

STARS CONFERENCE PLANNING TASKS

1	See a market	100 mags	Hant barace tam ting bin oth	OF 10 DOG 0: 10	100 mm tub	Ain smith	1100 EC 20
list & fvaluation	010 000 000 000 000 000 000 000 000 000	100 00 00 00 00 00 00 00 00 00 00 00 00	11 at as one	Count court	Deet affatte for deptemble for des sing	010000 MAXIM 010000 NICH (000) 000 MAX	Service Colores Service Colores Received
	7116 100.040 Aux. (705) 273 0318	TY TY	Sales	ore see and	Kita permas 18cmile 18cmile 18cmile	GALET BOOCH ANTICAM BACK (BY DES 2000	04 04 14 18 11 10 10 10 10 10 10 10 10 10 10 10 10
PUSHE 18	1980 954 K119	BEC one trad printery and award	Tract Saused From Saused Class and Class		DECORAGE AIR	Serie ese ferel series series affects series	Prit stand
MEASURE ME N.S.	100 200 100 100 100 100 100 100 100 100	bid see 1330 1800 1800 1800 1800 1800 1800	11 the part of the last of the	11 to 10 to	1500 1500 1500 1500 1500 1500 1500 1500	Pres and load socked comp (Jan 34 sock	ore to fact one south
SOFTWARE LEGISLIANG	the tre tract vytave tatemy stand	At the part free!	20 Biograf & Laston Day 1800 C. (2006) 174-7482	200 110 and	900 H/ bill 1907/40 1907/40	eek ees kird synoomis swoom per	Part by Place Heaves Heave Heave Heav
BUZZAB NESBUNCE ?	SUR BY HAD BODIS BYEN SU	DIG OF THE	11.00 EAVAGE 1500 (21.00 EAV 07.00	PACE ENDACIOS INI SISTEMBASE (301) 345-1070	I TANCO E BASSONO BELLOS (PAT BES OFFI (PAT	Dect (Amily Wass milling	
APPLICATION	100 TO REE! VIENT 107711 100	MAT 800 MATING SALE FAITH AND 8ALE MATING SALE MATING	11 mm 144444 12 mm 11 mm 12 mm	nes Checkstanas	pervell had	Ret- tres are beel surpose marre (2) se	0 200 100 100 100 100 100 100 100 100 10
PROCESSE BANKACHEES	1991/2007800 0 1/77 2001/200780 20 107 2007/200780 20 10 107 2071/200780 20 10000 00	THE PROPERTY OF THE PROPERTY O	Herr baland/Ab/1686 Herr (17% St. 6196 G. C.	A new days lines pri me gring 440 2046	peer at peer a contract (and a	M SE COME CLUTAL BURANCE FULL 211 AND	7147 2000 213 214 215 0000
***************************************	38/36	1 .	1	3	1		■ '

CONTACTS

•	ADA	D.G. FIRESMITH	MAGNAVOX	(219)429-4327
•	A1/ES	D.G. PRESTON G.B. WIGLE	I I T R I BOE I N G	(301)459-3711 (206)773-8591
•	FIRMWARE	A.J. MAHER	SINGER-KEARFOTT	(201)785-6607
•	SYSTEM ENGINEERING	A. McCULLOCH	HUGNES	(818)702-4271
•	REUSABLE SOFTWARE	R.D. KOLACKI	SPAWAR	(202)692-8484
•	EVOLUTIONARY ACQUISITION	E.C. BAUDER H. LYNESS	GTE GE	(617)449-2000 x5197 (315)456-1730
•	SOFTWARE QUALITY FACTORS	J.H. HEIL	111	(201)284-2946
•	METHODOLOGY	G. SUMRALL H. LEHRHAUPT	USA GE	(201)544-2670 (703)478-6278

TECHNICAL ADVISORY WORKING GROUP (TAMG) GUIDELINES

OBJECTIVES

ACT AS A SPONSOR AND CONVENOR FOR THE JOINT INDUSTRY, DOD AND

ACADEMIA TAWG

ASSIST CODSIA/JLC IN THE REVIEW AND ANALYSIS OF RAW COMMENTS

WITHIN THE SCOPE OF TAMG

ANALYZE ALTERNATIVES AND RECOMMEND SOLUTIONS TO CODSIA/JLC IN

THE AREA OF TAMG SCOPE

INFORM THE COMMUNITY AS TO EXISTING CONSTRAINTS AND PROVIDE

INTERIM GUIDANCE

- MONITOR AND REPORT ON THE REMOVAL OF CONSTRAINTS

ORGANIZATION

- SPONSOR

CHAIRPERSON'S DUTIES

VICE-CHAIRPERSON'S DUTIES

SECRETARY'S DUTIES

ASSOCIATED MEMBERS

CODSIA TAWG GUIDELINES

• PRODUCTS AND SERVICES

- TAWG CHARTER
- TAWG MEMBERSHIP LIST
- TAWG MAILING LIST
- MINUTES OF TAWG MEETINGS
- ISSUES AND CONCERNS ANALYSIS REPORT
- FORMAL RECOMMENDATIONS TO CODSIA AND JLC

CODSIA TAWG GUIDELINES

APPROACH

- MEETINGS
- WORKSHOPS
- STATUS REPORTING
- INTERIM CODSIA/JLC GUIDELINES
- COLLECTION OF COMMENTS
- ANALYSIS OF COMMENTS
 - COMMENTS
 - CONCERNS
 - SUBISSUES
 - ISSUES
- DEVELOPMENT OF ALTERNATIVES AND RECOMMENDATIONS
- REVIEW OF RECOMMENDATIONS
- FORMAL SUBMISSION OF RECOMMENDATIONS
- MONITORING AND ASSESSING ISSUE RESOLUTION

ADA TAVG

(79

SPONSOR: ACM SIGADA

DATE ORGANIZED: DECEMBER 1985

MEETINGS: DEC 1985 (BOSTON) -- CHARTER MEETING

● 10-12 FEB 1986 (CRYSTAL CITY) -- HDBK-287 ISSUES REVIEW

D FEB 1986 (LOS ANGELES) -- CHARTER

MEMBERSHIP:

PLANS:

INITIAL INPUTS HDBK-287 -- SDP ADA CRITERIA ACCOMPL ISHMENTS:

INITIAL ADA REV A GUIDANCE

FIRMWARE TAWG

SPONSOR: AIA ECSC

DATE ORGANIZED:

MEMBERSHIP:

PLANS: APR 1986 -- KICKOFF MEETING

AI/ES TAM

W TO

SPONSOR: EIA 6-34

DATE ORGANIZED:

MEETINGS:

16-20 SEP 1985 (ST LOUIS) -- EIA G-34 WORKSHOP AI/ES ISSUE STRUCTURE ANALYSIS

MEMBERSHIP:

PLANS:

AI/ES ISSUES TAXONOMY (EIA 6-34 WORKSHOP REPORT) ACCOMPLISHMENTS: 0

REUSABLE SOFTWARE TAMG

SPONSOR: STARS APPLICATION AREA

DATE ORGANIZED:

24-27 MAR 1986 (OXON HILL MD) -- STARS APPLICATION WORKSHOP

MEMBERSHIP:

MEETINGS:

PLANS:

COMMERCIAL AND REUSABLE SOFTWARE ISSUE (26)

SUBISSUES:

ADDDAWAI	
ACENICA	
PROFILDING	
.26-1	T 07.

26-2 TOP-DOWN HIERARCHICAL DESIGN REQUIREMENTS

26-3 SS&P MANUAL REQUIREMENTS

26-4 LC MANAGEMENT REQUIREMENTS

26-5 DATA RIGHTS

26-7 "OFF-THE-SHELF" -VS- "COMMERCIALLY AVAILABLE"

PDR ENFORCEMENT OF SOFTWARE REUSABILITY AND STANDARDIZATION 26-8 REV A

REQUIREMENT FOR "RATIONAL USE OF REUSABLE SOFTWARE" 26-9

APPROACH:

RANDOM COMMENTS -VS- STRUCTURED REVIEW APPROACH

DEVELOPMENT OF ADDITIONAL ISSUES AND RECOMMENDATIONS

DISPOSITION KEY:

- ACCEPTED/RESOLVED

- REJECTED

REV A - TO BE CONSIDERED FOR REV A

SECURITY CL	ASSIFICATION	OF THIS PAGE

REF JRT DOCUMENTATION PAGE Form Approved OMB No. 0704-0188							
1a. REPORT SECURITY CLASSIFICATION 1b. RESTRICTIVE MARKINGS UNCLASSIFIED							
2a. SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT						
2b. DECLASSIFICATION / DOWNGRADING SCHEDU	Approved for public release; distribution unlimited.						
4. PERFORMING ORGANIZATION REPORT NUMBE	5. MONITORING	ORGANIZATION R	EPORT NU	MBER(S)			
NRL Publication 0120-5150 6a. NAME OF PERFORMING ORGANIZATION 6b. OFFICE SYMBOL 7a. NAME OF MONITORING ORGANIZATION							
6a. NAME OF PERFORMING ORGANIZATION	6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MO	ONITORING ORGA	NIZATION			
Naval Research Laboratory	Code 5150	Naval Research Laboratory					
6c. ADDRESS (City, State, and ZIP Code) 7b. ADDRESS (City, State, and ZIP Code)							
Washington DC 20375-5000 Washington DC 20375-5000							
8a. NAME OF FUNDING / SPONSORING ORGANIZATION (If applicable) SPAWAR 8b. OFFICE SYMBOL 9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER (If applicable)							
8c. ADDRESS (City, State, and ZIP Code) 10 SOURCE OF FUNDING NUMBERS							
SPAWAR 613 PROGRAM PROJECT TASK WORK UNIT ACCESSION NO N							
31-2239							
11. TITLE (Include Security Classification) Software Technology for Adaptable Reliable Systems 12. PERSONAL AUTHOR(S)							
13a. TYPE OF REPORT 13b TIME COVERED 14. DATE OF REPORT (Year, Month, Day) 15 PAGE COUNT Conference Proceedings FROM 24 Mar to 27 Mar 368							
16 SUPPLEMENTARY NOTATION							
17 COSATI CODES	18 SUBJECT TERMS	Continue on reverse	e it necessary and	d identify t	by block number)		
FIELD GROUP SUB-GROUP							
19 ABSTRACT (Continue on reverse if necessary and identify by block number) Conference Proceedings for Workshop on application systems and reusability (24-27 March 1986).							
20 DISTRIBUTION / AVAILABILITY OF ABSTRACT 21 ABSTRACT SECURITY CLASSIFICATION							
20 DISTRIBUTION / AVAILABILITY OF ABSTRACT MUNCLASSIFIED/UNLIMITED SAME AS R	PT DTIC USERS	UNCLASSI	FIED				
22a. NAME OF RESPONSIBLE INDIVIDUAL Elizabeth Wald 22b. TELEPHONE (Include Area Code) (202) 767-3040 22c OFFICE SYMBOL Code 5150							
DD Form 1473, JUN 86 Previous editions are obsolete. SECURITY CLASSIFICATION OF THIS PAGE							

Ċ